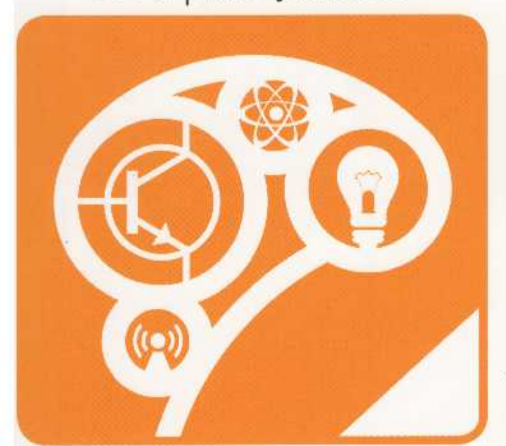
N°24 | MAI - JUIN 2018



HACKABLE MAGAZINE

DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

France MÉTRO. : 7,90 € – CH : 13 CHF – BEL/LUX/PORT.CONT : 8,90 € – DOM/TOM : 8,50 € – CAN : 14 $ CAD

REPÈRE

RGB, Péritel, Composite, S-Video, VGA, YPbPr... Tout ce que vous devez savoir sur la vidéo analogique

p. 30

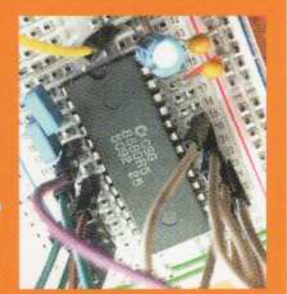


NOUVELLE RUBRIQUE

RETRO TECH

Créez un lecteur de chiptunes en pilotant la puce sonore d'un Commodore 64 avec une carte Arduino

p. 72



ESP32 / IDE

Développez efficacement vos croquis Arduino pour l'ESP32 avec l'ESP-IDF

p. 18

PI / MESURE

Mesurez simplement une tension avec votre Pi grâce à Python et un ADC SPI

p. 64

Domotique / Télécommande / ESP8266 / Wifi

Contrôlez votre éclairage d'intérieur en Wifi

p. 40

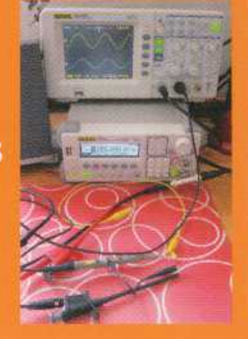
- Créez des ambiances et scénarios avec vos luminaires
- Interfacez une télécommande avec l'ESP8266
- Intégrez le contrôle de l'éclairage dans un navigateur



RASPBERRY PI / USB

Connectez vos oscilloscopes et générateurs de fonctions Rigol en USB pour automatiser vos mesures

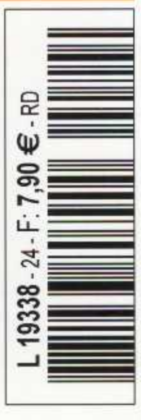
p. 04



8 BITS / Z80

Ajoutons un port série à notre ordinateur 8 bits Z80 et utilisons-le avec un petit programme en C

p. 88



30 JUIN - 01 JUILLET 2018

CITÉ DES SCIENCES ET DE L'INDUSTRIE

16

NUIT DU HACK

HTTP://WWW.NUITDUHACK.COM

~ ÉDITO ~



Faut-il avoir peur de la voiture autonome ?

L'accident mortel récent provoqué par un véhicule autonome Uber, il y a quelques semaines, a attisé une crainte déjà présente dans l'inconscient collectif. Je ne parle pas celle découlant uniquement de l'inconfort à laisser un ordinateur conduire un véhicule. Non, la problématique est, à mon sens, plus vaste et plus profonde que cela.

Saviez-vous qu'au début du siècle dernier, cette crainte existait déjà, dans un tout autre domaine (ou presque) ? Connaissez-vous le terme « liftier » ? Non ? Ceci n'est pas étonnant, il n'y en a presque plus. Un liftier est une personne affectée au contrôle d'un ascenseur, son « opérateur » en quelque sorte.

Au début des années 1900, l'idée même de laisser une machine fonctionner seule alors qu'elle transportait des personnes dans une boite suspendue au bout d'un câble, à quelques 200 mètres de haut (70 étages) était terrifiante. Rapidement pourtant la technologie était prête et la sécurité assurée. Mais le syndicat des liftiers ne voyait pas les choses de la même manière, préférant, sans surprise, insister sur l'importance d'une présence humaine aux commandes, au point d'atteindre en 1945 un point de basculement : une grève générale des liftiers, provoquant des blocages importants et un absentéisme catastrophique dans les bureaux des gratte-ciels.

Le mécontentement seul ne suffit pas à faire changer les mentalités, mais ce fut incontestablement l'élément déclencheur, le moment où la peur avait subitement une bonne raison d'être remise en cause. Les constructeurs durent également rendre les d'ascenseurs, non pas plus sûrs (ils l'étaient déjà), mais plus rassurants et plus conviviaux. Pourquoi pensez-vous que la « musique d'ascenseur » existe ?

Aujourd'hui, le fait qu'un humain puisse remplacer tous les systèmes de sécurité d'un ascenseur (ceux-là même qui le rendent inutile) est impensable. Mais en vérité, l'évolution aura tout de même pris près d'un demi-siècle et une bonne raison de réévaluer les choses. Mais ceci n'a pu arriver qu'après que la sécurité ait été pleinement assurée. Un ascenseur géré automatiquement est tout simplement plus sûr que s'il était commandé uniquement par un humain. Les quelques liftiers qui restent, dans les hôtels de luxe, ne font qu'utiliser l'ascenseur comme vous et moi, rien de plus.

La machine se doit donc d'être plus performante et c'est précisément le problème flagrant de l'accident Uber. Un humain n'aurait sans doute pas non plus pu éviter l'accident. Mais justement, là n'est pas la question. On ne demande pas à une machine d'être aussi sûre, fiable, et efficace que nous. On lui demande de l'être davantage.

Une technologie doit nous surpasser pour être acceptée, et c'est peut-être, là aussi, une façon de jouer avec le feu...



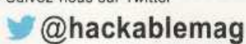
Hackable Magazine
est édité par Les Éditions Diamond



10, Place de la Cathédrale - 68000 Colmar
Tél. : 03 67 10 00 20 – Fax : 03 67 10 00 21
E-mail : lecteurs@hackable.fr
Service commercial : cial@ed-diamond.com
Sites : https://www.hackable.fr/
https://www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Responsable publicité : Valérie Fréchard, Tél. : 03 67 10 00 27 v.frechard@ed-diamond.com
Service abonnement : Tél. : 03 67 10 00 20

Impression : pva, Landau, Allemagne
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Abomarque : 09 53 15 21 77
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution,
N° ISSN : 2427-4631
Commission paritaire : K92470
Périodicité : bimestriel
Prix de vente : 7,90 €







~ À PROPOS DE HACKABLE... ~

HACKS, HACKERS & HACKABLE

Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un abus de langage médiatisé que de confondre « pirate informatique » et « hacker ». Le nom de ce magazine a été choisi pour refléter cette notion de **bidouillage créatif** sur la base d'un terme utilisé dans sa définition légitime, véritable et historique.

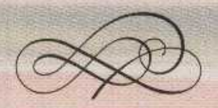
~ SOMMAIRE ~





AUTOMATISEZ VOS MESURES EN UTILISANT L'USB

Jean-Baptiste Vioix
Maître de conférences à l'IUT de Dijon-Auxerre - Département Réseaux et Télécoms



La marque Rigol est bien connue des hackers du monde entier. Elle propose des appareils de laboratoire pour l'électronique (oscilloscopes, générateurs...) ayant un prix accessible pour les amateurs. Dans cet article, nous allons présenter un oscilloscope et un générateur basses fréquences de ce fabricant, pilotables par le port USB pour automatiser les mesures. Cette automatisation va être utilisée (via Python) pour écrire un programme permettant de mesurer et tracer le diagramme de Bode d'un montage.



Fig. 1 : Les appareils en situation avec un filtre RC câblé « en l'air ».

Les appareils utilisés sont tous les deux de la marque Rigol. Cette marque totalement inconnue il y a une dizaine d'années est maintenant très présente dans le milieu de la bidouille. Son très bon rapport qualité-prix (qui produit même des oscilloscopes pour Agilent) n'y est pas étranger.

1. PRÉSENTATION DES APPAREILS RIGOL

1.1 Oscilloscope numérique DS1052E

Pour environ 300€ l'oscilloscope numérique DS1052E est l'un des rares oscilloscopes numériques accessibles à l'amateur en électronique.

La base de temps est réglable de 5 ns à 50 s par division pour une bande passante de 50 MHz. Les signaux les plus rapides visibles à l'oscilloscope seront donc représentés sur 4 carreaux, ce qui permet une lecture correcte. La base de temps la plus lente pourra être utilisée pour observer des variations lentes comme par exemple des capteurs de température ou de courant.

Les signaux acquis par les deux voies peuvent être combinés dans une troisième voie (MATH) à l'aide des opérateurs arithmétiques usuels. L'appareil propose aussi le calcul de la transformée de Fourier de l'une des deux voies (au choix) via le même menu (il est possible de choisir différentes fenêtres de pondération).

L'appareil est équipé de deux prises USB : une femelle A en façade qui permet de stocker des captures d'écran (ou des données brutes) sur une clé USB et une femelle B qui permet de contrôler l'oscilloscope depuis un PC.

L'une des rares critiques que l'on peut faire pour ce prix est que le ventilateur est un peu bruyant.

Pour 350€, Rigol propose le DS1102E qui est le même oscilloscope avec une bande passante étendue à 100 MHz. Les hackers les plus intrépides peuvent aussi trouver un firmware sur le Web permettant d'étendre la bande passante du 1052 à 100 MHz. Évidemment, cette modification invalidera instantanément la garantie de l'appareil...

1.2 Générateur basses fréquences DG1022

Après l'oscilloscope, le second appareil le plus utile dans un laboratoire d'électronique (surtout en électronique analogique) est le générateur de

fonctions ou générateur basses fréquences. Cet appareil permet de générer différents signaux périodiques (au minimum sinusoïdaux, triangulaires, rectangulaires) à différentes tensions (avec ou sans décalage par rapport à 0).

Le DG1022 regroupe toutes ces fonctions avec d'autres formes d'ondes (et la possibilité d'en charger des nouvelles depuis le port USB en façade) et surtout la présence de deux sorties totalement indépendantes. La fréquence du signal généré peut aller jusqu'à 20 MHz. À cela s'ajoutent plusieurs modulations : AM, FM, FSK... Le tout pour 320€.

Comme pour l'oscilloscope, une version supérieure existe : le DG1022A qui coûte environ 450 € pour une fréquence maximale de 25 MHz. Les derniers MHz sont chers ! Par contre, je n'ai pas trouvé trace d'un firmware permettant de récupérer les derniers MHz manquants d'un DG1022.

Les deux appareils peuvent être couplés pour que le générateur « rejoue » une séquence capturée par l'oscilloscope. Il est ainsi possible de travailler sur des séquences numériques que l'on souhaite décoder avec un microcontrôleur. Pour l'instant, je n'ai pas encore utilisé cette option.

2. CONTRÔLE DES APPAREILS PAR PC

Les appareils étant alimentés et reliés au port USB du PC (éventuellement via un hub), ils sont allumés. La commande **dmesg** permet ensuite de vérifier que les liaisons USB sont montées.

```
[100569.628422] usb 8-2.2: new full-speed USB device number 25 using xhci_hcd
[100573.861035] usb 8-2.2: New USB device found, idVendor=0400, idProduct=09c4
[100573.861038] usb 8-2.2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[100573.861040] usb 8-2.2: Product: DG1000 SERIES
[100573.861042] usb 8-2.2: Manufacturer: Rigol Technologies
[100573.861044] usb 8-2.2: SerialNumber: DG1**********
[100573.948098] usb 8-2.3: new full-speed USB device number 26 using xhci_hcd
[100574.076498] usb 8-2.3: New USB device found, idVendor=1ab1, idProduct=0588
[100574.076501] usb 8-2.3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[100574.076504] usb 8-2.3: Product: DS1000 SERIES
[100574.076505] usb 8-2.3: Manufacturer: Rigol Technologies
[100574.076507] usb 8-2.3: SerialNumber: DS1**********
```

La commande **lsub** confirme que tout est bien à sa place...

```
Bus 008 Device 026: ID 1ab1:0588 Rigol Technologies DS1000 SERIES
Bus 008 Device 025: ID 0400:09c4 National Semiconductor Corp. Rigol Technologies
DG1022 Arbitrary Waveform Generator
```

2.1 Interface de programmation VISA

Les instruments actuels sont très souvent compatibles avec l'interface de programmation VISA (*Virtual Instrument Software Architecture*). Les instruments Rigol sont compatibles et d'après la documentation du constructeur, il suffit d'installer le pilote VISA de National Instrument pour pouvoir contrôler les appareils.



2.1.1 Installation de VISA

Les éléments logiciels sont distribués par NI sur la page **http://www.ni.com/download/ni-visa-5.4.1/4629/en/**. Il faut ensuite télécharger le fichier `NI-VISA-5.4.1.iso`. Il est nécessaire de créer un compte chez NI pour pouvoir avoir accès au lien de téléchargement. NI annonce clairement la couleur lors de la création du compte, je ne peux que vous conseiller d'utiliser une adresse mail « poubelle » réservée à ce type d'usage...

> **Extrait des conditions de NI**
>
> **J'accepte que mes données personnelles soient recueillies, traitées et utilisées par NI selon les termes de sa déclaration de confidentialité.**
>
> **Je recevrai périodiquement des e-mails concernant les produits et services NI, et pourrai modifier mes préférences de communication à tout moment.**

L'image disque téléchargée est ensuite montée avec l'utilitaire du système (sur Ubuntu et dérivées l'application Monteur d'images disque est utilisée). Sinon une ligne de terminal est efficace aussi : `mount -o loop NI-VISA-5.4.1.iso /media/cdrom`.

Normalement seules les distributions Red Hat (et dérivées) sont reconnues par NI. Toutefois dans le fichier expliquant l'installation, il est proposé d'utiliser l'option `no-deps` et donc la commande `sudo ./INSTALL --nodeps` pour les autres distributions. Un kilomètre de licence s'affiche ensuite et l'installation échoue :-(Il va donc falloir trouver une autre solution.

Il existe une implémentation libre de VISA : LibreVISA, mais le projet n'est pas mis à jour depuis plusieurs années.

2.1.2 Installation de PyVISA-py

Puisque la solution propriétaire recommandée par le fabricant a échoué, il est temps de se pencher sur les solutions libres ! Pour contrôler des appareils de mesure avec Python, il est possible d'utiliser `pyvisa` qui nécessite un pilote VISA opérationnel ou `pyvisa-py` qui est une implémentation complète en Python de VISA. Les paquets `pyserial` et `pyusb` sont aussi nécessaires, les communications VISA pouvant se faire aussi bien en RS232 qu'en USB. L'installation se fait avec la ligne suivante :

```
sudo pip3 install pyserial pyusb pyvisa-py
```

3. PRISE EN MAIN DE LA LIBRAIRIE

3.1. Premier programme

Pour utiliser l'interface VISA, il faut commencer par importer le paquet `visa`. Ensuite, une librairie doit être spécifiée avec la fonction `ResourceManager`, pour



l'implémentation Python la chaîne @py doit être utilisée. Il ne reste plus qu'à obtenir une liste des instruments connectés avec la méthode list_ressources.

```
import visa
rm = visa.ResourceManager('@py')
instruments = rm.list_resources()
print(instruments)
```

La liste des périphériques compatibles VISA s'affiche et on retrouve l'oscilloscope et le GBF dans la liste avec les identifiants DS et DG suivis ensuite de leurs numéros de série. Une recherche dans la liste permet ensuite d'associer chaque appareil à un objet de la classe USBInstrument.

```
for _ in instruments:
 if 'DS' in _:
  oscilloscope = rm.open_resource(_)
 if 'DG' in _:
  generator = rm.open_resource(_)
```

Ensuite, il est préférable de réinitialiser les différents appareils avant d'envoyer d'autres commandes en envoyant une commande *RST avec la méthode query de la classe USBInstrument.

```
generator.query("*RST")
oscilloscope.query("*RST")
```

3.2. Contrôle du générateur

La documentation du constructeur présente l'ensemble des instructions possibles pour contrôler le générateur. Dans la suite de ce document, nous n'en utiliserons que deux :

- OUTP qui permet de contrôler l'état de la sortie (activée ou non) ;
- APPL:SIN est utilisée pour générer un signal sinusoïdal caractérisé par sa fréquence, son amplitude crête-à-crête et son décalage (par rapport à zéro).

Les commandes sont simples à utiliser, ce sont des chaînes de caractères que l'on envoie à l'appareil via la méthode query :

```
generator.query("APPL:SIN 440,1.0,0.0")
generator.query("OUTP ON")
```

Le générateur produit maintenant une sinusoïde de fréquence 440 Hz avec une tension crête-à-crête de 1 V et un décalage nul.

3.3 Contrôle de l'oscilloscope

Les oscilloscopes numériques sont souvent équipés d'un mode de réglage automatique (*autoset*) qui permet souvent de « dégrossir » les réglages. Il très simple d'activer le mode automatique via python :

```
oscilloscope.query(":AUTO")
```

Un petit claquement de relais se produit comme lorsque le mode automatique est activé manuellement. Le réglage automatique affiche les deux voies (même si rien n'est branché sur la seconde voie). Les mesures automatiques peuvent ensuite être utilisées pour relever la fréquence et la tension du signal de chaque voie. Pour l'instant, seule la première voie est utilisée, deux lignes sont nécessaires pour lire les valeurs.

```
print(u"Fréquence : %f Hz"%(float(oscilloscope.query(":MEAS:FREQ? CHAN1"))))
print(u"Tension (CàC) : %f V"%(float(oscilloscope.query(":MEAS:VPP? CHAN1"))))
```

La méthode **query** retourne les données lorsque la commande est une requête (généralement terminée par ?). Les données retournées sont des chaînes de caractères, elles doivent donc être converties en flottant avant des calculs (pour un simple affichage ce n'était pas nécessaire, mais autant prendre de bonnes habitudes tout de suite...).

3.4 Accès non root

L'accès direct aux périphériques USB n'est pas possible pour l'utilisateur de base. Il est donc nécessaire d'utiliser **sudo** pour exécuter le script.

Il est possible de donner l'accès direct aux périphériques USB en ajoutant une règle pour le gestionnaire de périphérique **udev**. Les opérations à réaliser sont les suivantes (en super-utilisateur) :

- créer un groupe **usbusers** et ajouter l'utilisateur à ce groupe (soit en ligne de commandes, soit avec l'interface graphique...) ;
- créer un fichier **00-usbusers.rules** dans le répertoire **/etc/udev/rules.d** ;
- dans ce fichier, ajouter les lignes suivantes :

```
SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_device", MODE="0664", GROUP="usbusers"
```

L'utilisateur doit ensuite se déconnecter et se reconnecter pour qu'il puisse accéder aux périphériques USB sans utiliser la commande **sudo**.

4. CONSTRUCTION DU DIAGRAMME DE BODE

4.1 Présentation du diagramme de Bode

Le diagramme de Bode est une représentation graphique utilisée en électronique analogique pour caractériser le comportement d'un montage en fonction de la fréquence du signal d'entrée. Il est composé de deux graphes :

- Le diagramme du gain qui représente l'évolution du gain (ou de l'atténuation) du montage en fonction de la fréquence. L'évolution de la tension de sortie en fonction de la tension d'entrée est exprimée en décibels (dB) avec la formule de calcul suivante :

$$G_{dB} = 20 \log \left(\frac{V_S}{V_E} \right)$$

  Une amplification est donc représentée par des valeurs positives, une atténuation par une valeur négative. L'utilisation des décibels permet d'avoir des valeurs numériques « compactes », ainsi un gain de 10000 est représenté par +80 dB.
- Le diagramme de phase représente le décalage temporel entre la sortie et l'entrée du montage. Comme le signal d'entrée est une fonction sinusoïdale, le déphasage est exprimé sous la forme d'un angle en radians. Ainsi, un décalage d'une demi-période est représenté par -π.

En pratique, le diagramme de phase est moins utilisé que le diagramme de gain.

4.2 Parcours des fréquences

Les diagrammes de Bode utilisent une échelle logarithmique pour les fréquences. Il est donc nécessaire de parcourir l'étendue des fréquences de manière non linéaire, sinon les points ne seront pas répartis de manière homogène sur le diagramme.

Dans le paquet **numpy**, la fonction **logspace** retourne un ensemble de valeurs réparties de manière logarithmique sur une base (qui est 10 par défaut), les bornes étant des puissances de la base. Le dernier paramètre obligatoire est le nombre de valeurs souhaitées. Pour construire un diagramme de Bode entre 100 Hz et 100 kHz avec un total de 30 points la fonction sera appelée de la manière suivante **logspace(2,5,30)**.

```
NB = 30
results = np.zeros((NB, 3))
generator.query("OUTP ON")
for idx, f in enumerate(np.logspace(2, 5, NB)):
  generator.query("APPL:SIN %f,3.0,0.0" % (f,))
  oscilloscope.query(":AUTO")
  time.sleep(5)
  results[idx, 0] = float(oscilloscope.query(":MEAS:FREQ? CHAN1"))
  results[idx, 1] = float(oscilloscope.query(":MEAS:VRMS? CHAN1"))
  results[idx, 2] = float(oscilloscope.query(":MEAS:VRMS? CHAN2"))
```

À chaque itération de la boucle, la nouvelle valeur de fréquence est envoyée au GBF puis l'oscilloscope est reconfiguré automatiquement. Une pause de quelques secondes est nécessaire pour que la fonction *autoset* se termine avant de demander les mesures de fréquence et de tensions (ici il s'agit de la tension efficace *Root Mean Square* en anglais).

4.3 Calcul et affichage de l'évolution du gain

Les données étant stockées dans un tableau **ndarray**, la lecture des fréquences se fait en accédant à la première colonne, le calcul du gain se fait à partir de la deuxième et de la troisième colonne.

L'affichage avec un axe des abscisses logarithmique utilise la fonction **semilogx** de **matplotlib**. Les autres éléments permettent de mettre en forme le diagramme :

```
plt.figure(figsize=(15, 10))
plt.semilogx(results[:, 0], 20*np.log10(results[:, 2]/results[:, 1]), '-o')
plt.grid(True, 'minor')
plt.grid(True, 'major')
plt.ylabel('Amplitude [dB]')
plt.xlabel(u'Fréquence [Hz]')
plt.title("Diagramme de Bode du filtre")
plt.savefig("TeBNC.pdf")
plt.show()
```

En utilisant un té BNC les deux entrées de l'oscilloscope sont reliées à la sortie du générateur. Le diagramme de Bode obtenu est affiché dans la figure 2 ci-contre.

En théorie, le diagramme de Bode serait parfaitement plat. Les légères ondulations qui sont de l'ordre de +/- 0,01 dB sont principalement dues à la précision des mesures automatiques de l'oscilloscope.

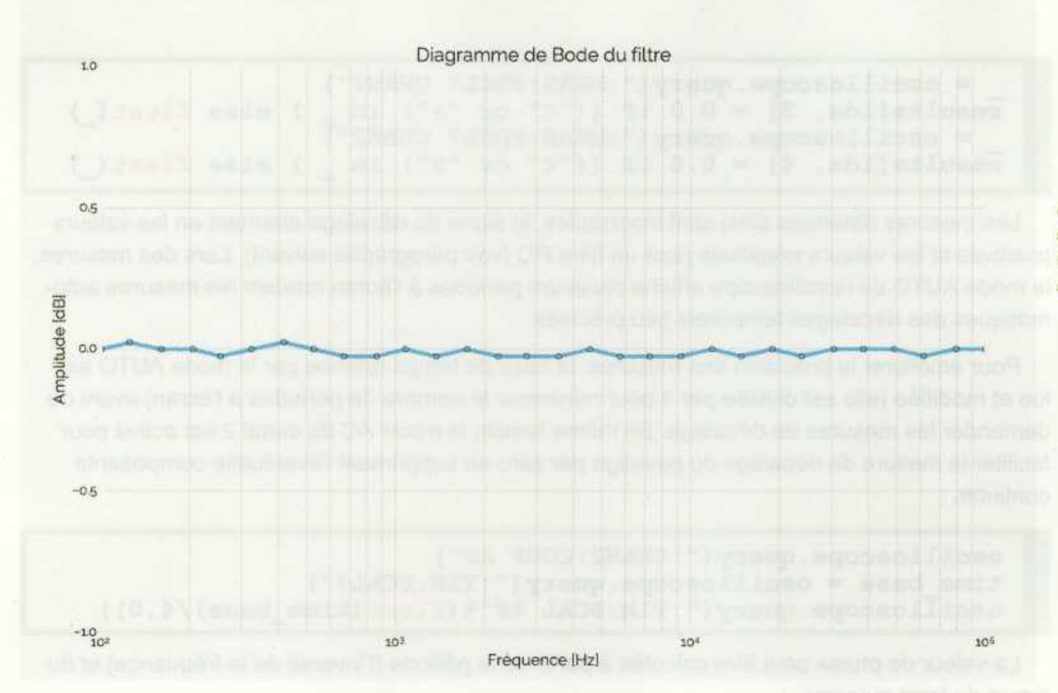


Fig. 2 : Diagramme de Bode du té BNC.

4.4 Sauvegarde des données

En plus de l'enregistrement de la figure du diagramme de Bode, les données sont sauvegardées dans un fichier CSV pour pouvoir éventuellement être réutilisées ultérieurement. Le module **csv** de Python propose toutes les fonctions utiles pour manipuler les fichiers CSV. Afin de pouvoir retrouver le rôle des différentes colonnes dans le fichier, la première ligne contient le nom des différentes variables mesurées. Le caractère **#** est utilisé comme marqueur pour la signaler comme une entête.

```
with open("Mesures.csv", "w") as f:
 writer = csv.writer(f, delimiter=',')
 writer.writerow(["#Freq. (Hz)", "In (Vrms)", "Out (Vrms)"])
 writer.writerows(results)
```

4.5 Évolution de la phase

Dans un diagramme de Bode complet, l'évolution de la phase en fonction de la fréquence est aussi tracée.

La phase ne peut pas être directement mesurée avec l'oscilloscope. Il est par contre possible de mesurer l'avance ou le retard d'une voie par rapport à l'autre. À partir de ces informations, il doit être possible de reconstruire la phase.

Les valeurs de décalage du front montant et du front descendant du canal 2 par rapport au canal 1 sont mesurées. Les mesures ne peuvent pas être directement converties en valeurs flottantes, car l'oscilloscope peut retourner des chaînes du type **<1.0us**. La conversion nécessite donc un test sur la présence des symboles **<** ou **>** pour éviter une exception. Le tableau **results** a été agrandi en conséquence (à 5 colonnes) lors de sa construction.

```
_ = oscilloscope.query(":MEAS:PDEL? CHAN2")
results[idx, 3] = 0.0 if (("<" or ">") in _ ) else float(_)
_ = oscilloscope.query(":MEAS:NDEL? CHAN2")
results[idx, 4] = 0.0 if (("<" or ">") in _ ) else float(_)
```

Les mesures obtenues ainsi sont incorrectes, le signe du décalage alternant en les valeurs positives et les valeurs négatives pour un filtre RC (voir paragraphe suivant). Lors des mesures, le mode AUTO de l'oscilloscope affiche plusieurs périodes à l'écran rendant les mesures automatiques des décalages temporels peu précises.

Pour améliorer la précision des mesures, la base de temps retenue par le mode AUTO est lue et modifiée (elle est divisée par 4 pour minimiser le nombre de périodes à l'écran) avant de demander les mesures de décalage. En même temps, le mode AC du canal 2 est activé pour faciliter la mesure de décalage du passage par zéro en supprimant l'éventuelle composante continue.

```
oscilloscope.query(":CHAN2:COUP AC")
time_base = oscilloscope.query(":TIM:SCAL?")
oscilloscope.query(":TIM:SCAL %f"%(float(time_base)/4.0))
```

La valeur de phase peut être calculée à partir de la période (l'inverse de la fréquence) et du retard du front montant.

```
periods = 1.0/results[:, 0]
phases = -2.0*np.pi*(results[:, 3]/periods)
```

Il serait aussi possible de régler automatiquement la base de temps en utilisant la valeur de la fréquence. Cette solution n'a pas été mise en œuvre, le diagramme de phase n'étant pas la partie la plus utilisée du diagramme de Bode.

5. EXEMPLE D'UTILISATION : FILTRE RC

5.1. Éléments de théorie

Les filtres sont des montages électroniques dont le comportement dépend de la fréquence. Les plus simples sont réalisés avec une résistance et un condensateur. Selon le câblage, le filtre peut avoir l'un des deux comportements suivants :

- passe bas : le filtre laisse passer les signaux ayant une fréquence inférieure à une certaine valeur (la fréquence de coupure). Les signaux ayant une fréquence supérieure sont atténués, plus leur fréquence est élevée, plus ils seront atténués.
- passe haut : le comportement est le dual du précédent. Les signaux ayant une fréquence supérieure à la fréquence de coupure ne sont pas altérés. Sinon, l'atténuation est inversement proportionnelle à la fréquence (ainsi les signaux continus sont totalement atténués.)

Le câblage d'un filtre passe-bas est présenté dans la figure 2.

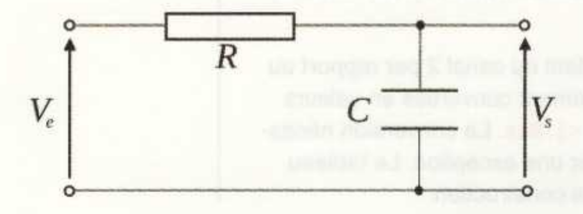


Fig. 3 : Schéma d'un filtre passe-bas.

La fréquence de coupure d'un filtre RC est donnée par la formule :

$$f_c = \frac{1}{2\pi . R.C}$$

avec R la résistance en Ω et C la capacité en Farads (F). Avec une résistance de 100kΩ et un condensateur de 1,5 nF, la fréquence de coupure théorique est de 1,061 kHz (à cette valeur le gain vaut -3 dB).

Avec le module **scipy.signal**, il est possible de calculer le diagramme de Bode théorique du filtre. Un filtre composé d'une résistance et d'un condensateur est modélisé par la fonction de Butterworth ; cette fonction est nommée **butter** dans le module. Elle utilise 2 paramètres : l'ordre du filtre (1 pour un filtre RC) et la pulsation de coupure (qui vaut ω=1/(R.C), car ω=2 π.f). Les valeurs de retour de la fonction (de type **ndarray**) sont :

- les pulsations pour lesquelles la fonction de transfert a été calculée,
- les valeurs (sous forme complexe) de la fonction de transfert à ces pulsations.

À partir de ces valeurs, les fréquences sont calculées (avec la relation ci-dessus) ainsi que le gain en décibels (à partir du module de la fonction de transfert).

```
b, a = signal.butter(1, 1.0/(100e3*1.5e-9), 'low', analog=True)
w, h = signal.freqs(b, a)
f_th = w/2*np.pi
g_th = 20*np.log10(abs(h))
```

Les fréquences et le gain ont été placés dans des variables notées **_th** puisqu'ils correspondent à des valeurs théoriques.

5.2 Mesures pratiques

5.2.1 Affichage du gain

À partir des éléments de programme présentés dans le chapitre précédent, le gain du filtre a été mesuré entre 100 Hz et 100 kHz ; 30 points ont été mesurés. Les valeurs ont été stockées dans le fichier **MesuresRC.csv**. Le programme de calcul théorique du filtre est complété pour lire ces données et les afficher sur un diagramme de Bode avec les données théoriques.

```
# Lecture des données
resultats = []
with open("MesuresRC.csv", "r") as f:
    reader = csv.reader(f, delimiter=',')
    next(reader, None) # pour supprimer la première ligne
    for line in reader:
        resultats.append([float(_) for _ in  line ])
results = np.array(resultats)
f_me = results[:, 0]
g_me = 20*np.log10(results[:, 2] / results[:, 1])
```

```
# Création de la figure
plt.figure(figsize=(15, 10))
plt.semilogx(f_th, g_th, '-')
plt.semilogx(f_me, g_me, '-o')
plt.grid(True, 'minor')
plt.grid(True, 'major')
plt.ylabel("Amplitude [dB]")
plt.xlabel("Fréquence [Hz]")
plt.title("Diagramme de Bode du filtre RC")
plt.legend(("Théorique","Pratique"))
plt.margins(0.1, 0.1)
plt.savefig("FiltreRC.png")
plt.show()
```

Le chargement des données est simple, il faut simplement passer la première ligne (avec la directive **next**) qui contient les noms de colonnes. Les données lues à partir des mesures sont notées avec **_me** (pour les différencier des données théoriques). Le gain du filtre est calculé à partir de la formule donnée dans la présentation théorique.

Ensuite, différentes fonctions de Matplotlib permettent de mettre le diagramme de Bode en forme. Pour la courbe représentant les données pratiques les points de mesure sont signalés par une marque ronde.

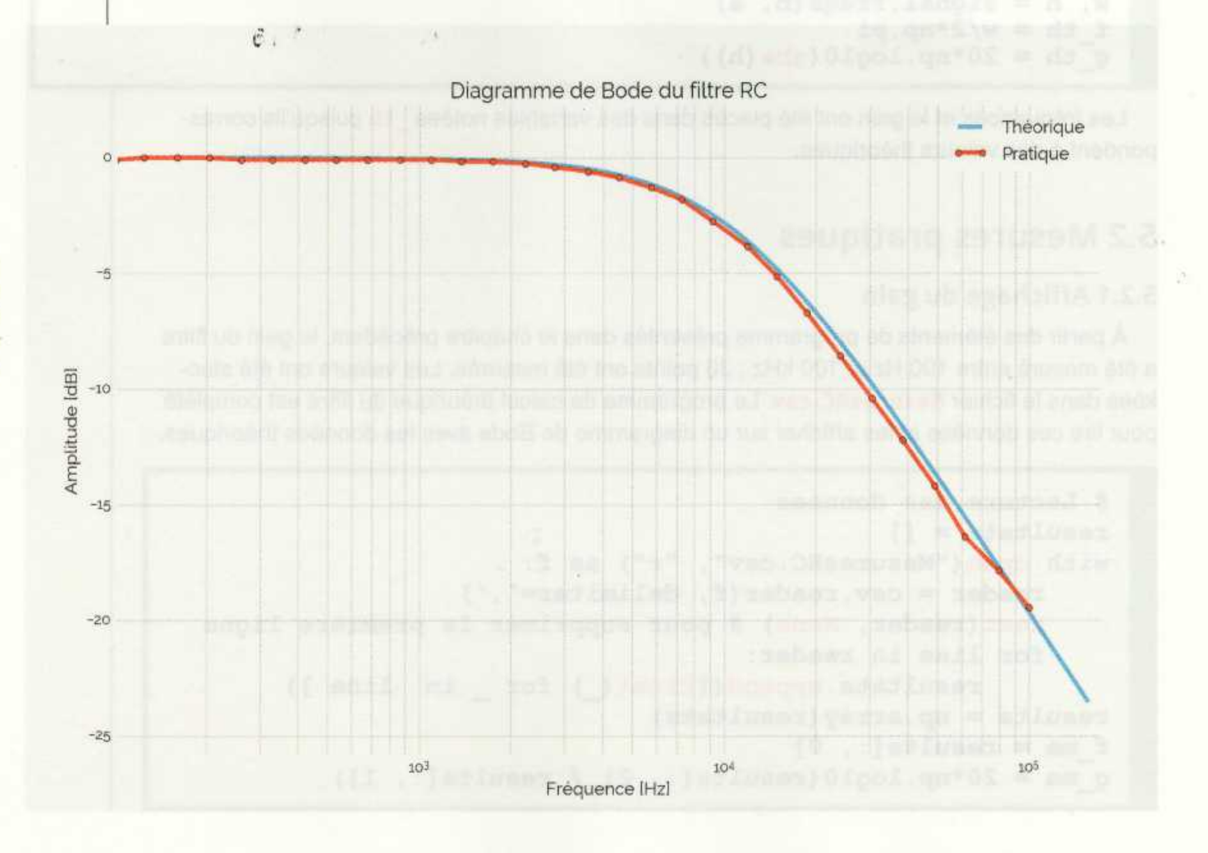


Fig. 4 : Diagramme de Bode (gain) pour un filtre RC.

La superposition sur le même graphique de la courbe théorique et des mesures permet de vérifier le comportement du filtre. Dans le cas de notre montage, les courbes sont presque confondues. Les valeurs des composants étaient très proches des valeurs théoriques : après mesure, la résistance fait exactement 100kΩ et le condensateur 1508 pF. Les résistances ont souvent une tolérance de +/-5 % et les condensateurs +/-10 %. En prenant en compte les cas extrêmes, la fréquence de coupure pratique du filtre pourrait être comprise entre 918 Hz et 1,24 kHz.

5.2.2 Affichage de la phase

Comme pour le gain, la phase théorique et la phase mesurée vont être affichées sur le même graphique. La phase théorique est calculée avec la fonction angle de Python.

```
ph_th = np.angle(h)
```

Comme nous l'avons présenté ci-dessus, la phase pratique est calculée à partir du retard avec le passage par zéro :

```
periods = 1.0/results[:, 0]
ph_me = -2.0*np.pi*(results[:, 3]/periods)
```

Les mêmes éléments graphiques sont repris pour l'affichage des deux courbes.

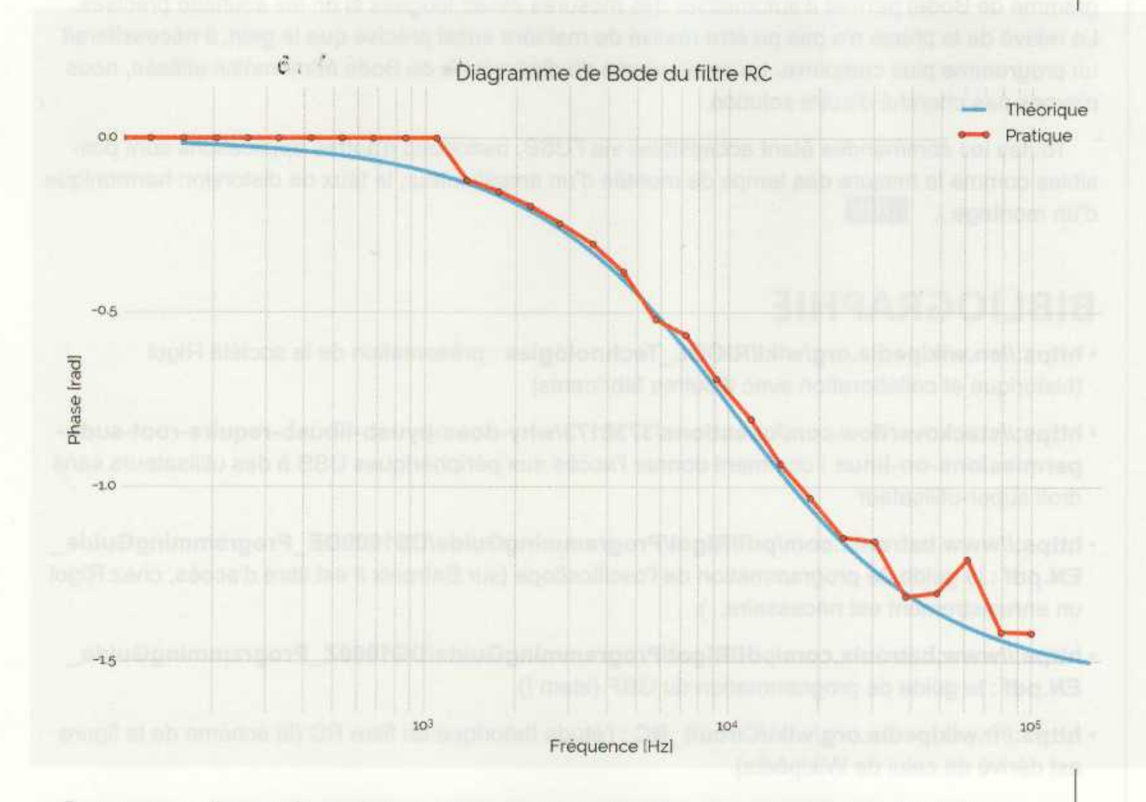


Fig. 5 : Diagramme de Bode (phase) pour un filtre RC.

Comme nous l'avons évoqué dans le paragraphe présentant la mesure de la phase, les mesures sont moins précises que pour le gain.



CONCLUSION

Après avoir fait une présentation rapide des deux appareils utilisés, nous avons montré qu'il est simple de les contrôler par le port USB. L'application proposée (le relevé automatique du diagramme de Bode) permet d'automatiser des mesures assez longues si on les souhaite précises. Le relevé de la phase n'a pas pu être réalisé de manière aussi précise que le gain, il nécessiterait un programme plus complexe. La partie phase du diagramme de Bode étant moins utilisée, nous n'avons pas cherché d'autre solution.

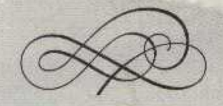
Toutes les commandes étant accessibles via l'USB, beaucoup d'autres applications sont possibles comme la mesure des temps de montée d'un amplificateur, le taux de distorsion harmonique d'un montage... **JBV**

BIBLIOGRAPHIE

- **https://en.wikipedia.org/wiki/RIGOL_Technologies** : présentation de la société Rigol (historique et collaboration avec d'autres fabricants)
- **https://stackoverflow.com/questions/3738173/why-does-pyusb-libusb-require-root-sudo-permissions-on-linux** : comment donner l'accès aux périphériques USB à des utilisateurs sans droit super-utilisateur
- **https://www.batronix.com/pdf/Rigol/ProgrammingGuide/DS1000DE_ProgrammingGuide_EN.pdf** : le guide de programmation de l'oscilloscope (sur Batronix il est libre d'accès, chez Rigol un enregistrement est nécessaire...)
- **https://www.batronix.com/pdf/Rigol/ProgrammingGuide/DG1000Z_ProgrammingGuide_EN.pdf** : le guide de programmation du GBF (idem !)
- **https://fr.wikipedia.org/wiki/Circuit_RC** : l'étude théorique du filtre RC (le schéma de la figure est dérivé de celui de Wikipédia)
- **https://github.com/jbvioix/mesure_usb** ou **https://gitlab.com/jbvioix/mesures_usb** : le dépôt contenant les fichiers associés à ce projet

ESP32 : DÉVELOPPEZ VOS CROQUIS ARDUINO SANS L'IDE ARDUINO

Denis Bodor



L'un des avantages des cartes ESP32, comme pour les ESP8266, est leur possibilité d'utilisation dans l'environnement Arduino. Grâce à cela, il est possible de bénéficier de tous les bienfaits de la plateforme tout en disposant de la simplicité d'utilisation d'Arduino. Ce n'est, bien entendu, pas le seul moyen de créer des programmes pour l'ESP32 puisqu'un environnement officiel existe. Entre les deux approches, une solution mitoyenne existe : utiliser l'environnement officiel Espressif, mais avec un composant Arduino.

Naturellement, on peut se demander quel est l'intérêt de se passer de l'interface de développement Arduino. Après tout, même s'il est relativement simpliste, il fait parfaitement bien son travail et permet de se concentrer sur l'essentiel, son projet. On ignore souvent les bienfaits de choses dont on a pas l'habitude (on les ignore aussi parfois justement quand on en a l'habitude, mais c'est un autre débat). La simplicité de l'IDE Arduino est à double tranchant. Certes, tout est accessible d'un simple clic, mais aucune fonction avancée n'est proposée : pas de gestion de révision intégrée, fonctions recherche/remplacement basiques, pas d'autocomplétion, mécanisme de débogage, etc.

Inversement, ce sur quoi le support ESP32 pour Arduino est construit, le SDK Espressif, ne propose aucun environnement de développement intégré, ce qui permet au développeur d'utiliser celui de son choix. Certains aiment des environnements complets et graphiques comme Eclipse, et d'autres, comme moi, affectionnent la bonne vieille solution qui a fait ses preuves : un bon éditeur de code comme Vim, quelques lignes de commandes et des outils précis (Git, `grep`, `make`, `sed`, Doxygen, etc.).

Espressif fourni une base de développement généralement appelée un SDK pour « *Software Development Kit* », mais qui, dans ce cas précis, est nommé IDF, pour « *IoT Development Framework* » et plus exactement ESP-IDF. Celui-ci vous permet de développer des programmes pour l'ESP32, mais aussi de choisir précisément quelles fonctionnalités vous souhaitez utiliser, aussi bien en termes de caractéristiques de la plateforme que concernant des facilités de mise au point de vos codes (débogage). Certaines fonctions ne sont disponibles que si leur utilisation est activée dans l'IDF. C'est le cas, par exemple, pour tout ce qui concerne la collecte d'informations sur les tâches en cours de fonctionnement. Ceci est tout simplement inaccessible via l'environnement Arduino puisque l'IDF intégré dans le support est une version « allégée » et configurée pour cet usage, et n'offre pas de personnalisation facilement accessible.

Chose qu'on ignore en utilisant simplement l'environnement Arduino pour ESP32, le support est architecturé de façon très intelligente de façon à minimiser les efforts des développeurs. On peut ainsi penser que le support ESP32 pour Arduino est une version spéciale de l'IDF dont une copie est faite, triturée et adaptée. Les choses sont un peu plus élégantes, car en réalité, le support Arduino (et je parle ici du « langage » et des bibliothèques propres à Arduino) est un élément complémentaire à l'IDF. Ce n'est donc pas simplement le support ESP32 qui est une option pour l'IDE Arduino, mais plutôt le support Arduino qui est un « greffon » pour l'IDF Espressif et le tout est « emballé » pour devenir une « extension » optionnelle pour l'environnement Arduino.

Ce qu'il faut retenir de tout cela, c'est tout simplement que l'ensemble est furieusement modulaire. Vous pouvez tout aussi bien ajouter le support ESP32 à votre environnement Arduino, ou ajouter le support Arduino à l'IDF Espressif. Dans un cas comme dans l'autre, un croquis « saveur Arduino » pourra être développé et fonctionnera sur votre carte ESP32, mais d'un côté vous utiliserez une « marmite » Arduino et de l'autre une « marmite » Espressif, selon vos préférences.

La marmite Arduino, vous la connaissez. Penchons-nous donc sur celle d'Espressif...

1. OPTION : COMPILER LE COMPILATEUR SUR PI 3

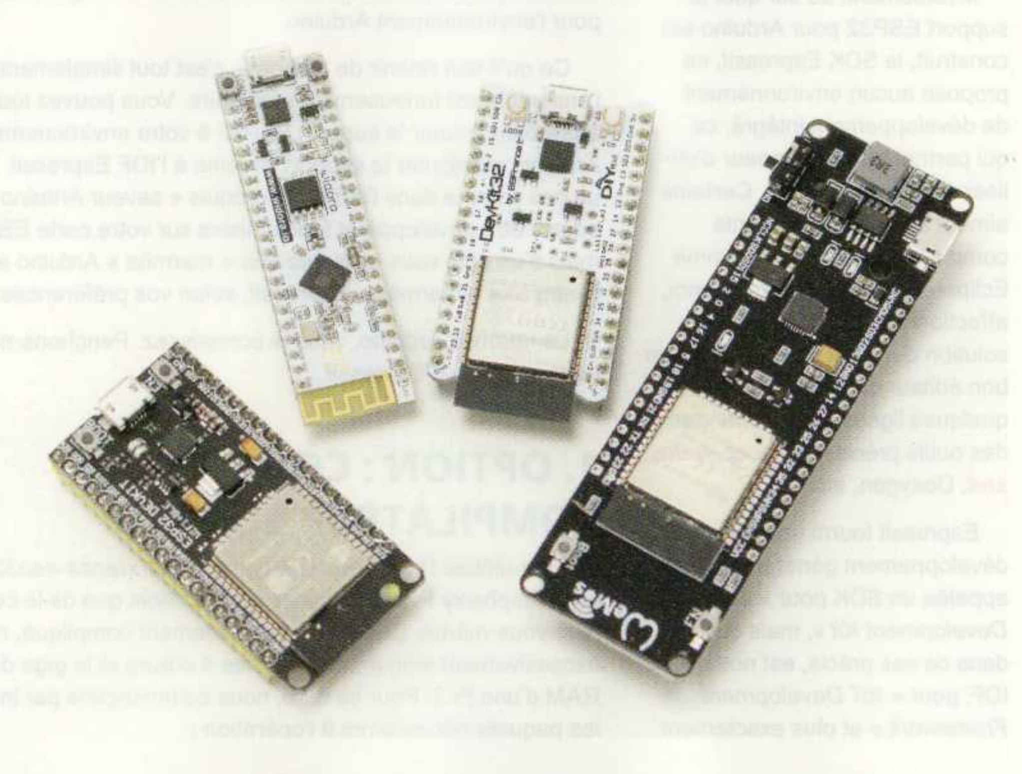
Pour utiliser l'indispensable compilateur *xtensa-esp32* sur une Raspberry Pi vous n'aurez d'autre choix que de le compiler vous-même. Ceci n'est pas réellement compliqué, mais excessivement lent, même avec les 4 cœurs et le giga de RAM d'une Pi 3. Pour ce faire, nous commençons par installer les paquets nécessaires à l'opération :

```
$ sudo apt-get install gawk gperf grep gettext \
python python-dev automake bison flex texinfo \
libncurses5-dev help2man libtool libtool-bin git
```

Puis nous créons un répertoire et y plaçons les éléments récupérés du dépôt officiel :

```
$ mkdir ESP32
$ cd ESP32
$ git clone -b xtensa-1.22.x https://github.com/espressif/crosstool-NG.git
Clonage dans 'crosstool-NG'...
remote: Counting objects: 28593, done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 28593 (delta 5), reused 17 (delta 5), pack-reused 28558
Réception d'objets: 100% (28593/28593), 16.01 MiB | 5.65 MiB/s, fait.
Résolution des deltas: 100% (16454/16454), fait.
Vérification de la connectivité... fait.
```

La construction repose sur un générateur de chaîne de compilation appelé Crosstool-NG. Cet outil utilise une configuration spécifique propre à une chaîne de compilation cible, et s'occupe de télécharger, configurer, et compiler les sources afin d'obtenir un environnement complet permettant de compiler des programmes à destination d'une plateforme donnée. Ce que vous obtenez en téléchargeant le compilateur d'Espressif pour l'ESP32 est le résultat de l'utilisation de Crosstool-NG avec une configuration spécifique à l'ESP32. Ici, plutôt que de télécharger quelque chose de clé en main, nous devons construire tout cela comme des grands.



La diversité des cartes et modules ESP32 ne cesse de croître et les prix ne cessent de baisser, au point de faire presque passer au second plan leurs prédécesseurs construits autour de l'ESP8266. Nous avons là une plateforme en plein développement qu'il ne faut clairement pas perdre de vue.

On commence donc par compiler Crosstool-NG lui-même :

```
$ cd crosstool-NG
$ ./bootstrap
Running autoconf...
Done. You may now run:
    ./configure

$ ./configure --enable-local
checking build system type... armv7l-unknown-linux-gnueabihf
checking host system type... armv7l-unknown-linux-gnueabihf
checking for a BSD-compatible install... /usr/bin/install -c
checking for grep that handles long lines and -e... /bin/grep
checking for egrep... /bin/grep -E
[...]
checking for library containing tgetent... none required
configure: overiding all of --prefix and the likes,
  because --enable-local was set
configure: creating ./config.status
config.status: creating Makefile

$ make install
  SED    'ct-ng'
  SED    'scripts/crosstool-NG.sh'
  SED    'scripts/saveSample.sh'
[...]
  SED    'docs/ct-ng.1'
  GZIP   'docs/ct-ng.1.gz'
```

Notez que ce dernier **make install** ne nécessite pas l'utilisation de **sudo**. En effet, nous avons utilisé **--enable-local** avec l'outil **configure**. Cette option configure les sources pour une utilisation de l'outil dans le répertoire courant, évitant ainsi toute installation sauvage et impossible à gérer ensuite (oui, **./configure && make && sudo make install** c'est mal, très très mal (et sale)).

Crosstool-NG est maintenant prêt à être utilisé et nous pourrions en principe utiliser **./ct-ng menuconfig** pour configurer manuellement le compilateur que nous souhaitons obtenir. Heureusement pour nous, une configuration par défaut existe et il nous suffit d'invoquer la commande suivante pour l'utiliser :

```
$ ./ct-ng xtensa-esp32-elf
  MKDIR config.gen
  IN    config.gen/arch.in
  IN    config.gen/kernel.in
[...]
WARNING! This sample may enable experimental features.
         Please be sure to review the configuration prior
         to building and using your toolchain!
Now, you have been warned!
Now configured for "xtensa-esp32-elf"
```

Comme le précise le message, cette configuration doit être considérée comme un « exemple » et devrait être vérifiée avant de construire et d'utiliser la chaîne de compilation. En réalité, étant donné qu'il s'agit d'une configuration standard en provenance d'Espressif, nous pouvons nous lancer sans attendre.

Attention cependant, cette étape est celle qui prendra **énormément** de temps et consommera presque toutes les ressources de votre Pi. Je vous recommande également de mettre en place un système pour refroidir votre Pi. Un simple petit radiateur en aluminium collé au SoC Broadcom a été bien insuffisant dans mon cas et j'ai été obligé de recycler un ventilateur dérobé sur une vieille carte mère de Pentium 4 pour éviter au système de s'arrêter net après 10 minutes de travail. Grâce à cette astuce, non seulement l'opération s'est bien déroulée, mais, en plus, la température du processeur (`/opt/vc/bin/vcgencmd measure_temp`) a rarement dépassé 50°C.

La construction de la chaîne de compilation sera déclenchée par :

```
$ ./ct-ng build
[INFO ]  Performing some trivial sanity checks
[INFO ]  Build started 20180129.124928
[INFO ]  Building environment variables
[...]
[INFO ]  Build completed at 20180129.150532
[INFO ]  (elapsed: 96:19.24)
[INFO ]  Finishing installation (may take a few seconds)...
```

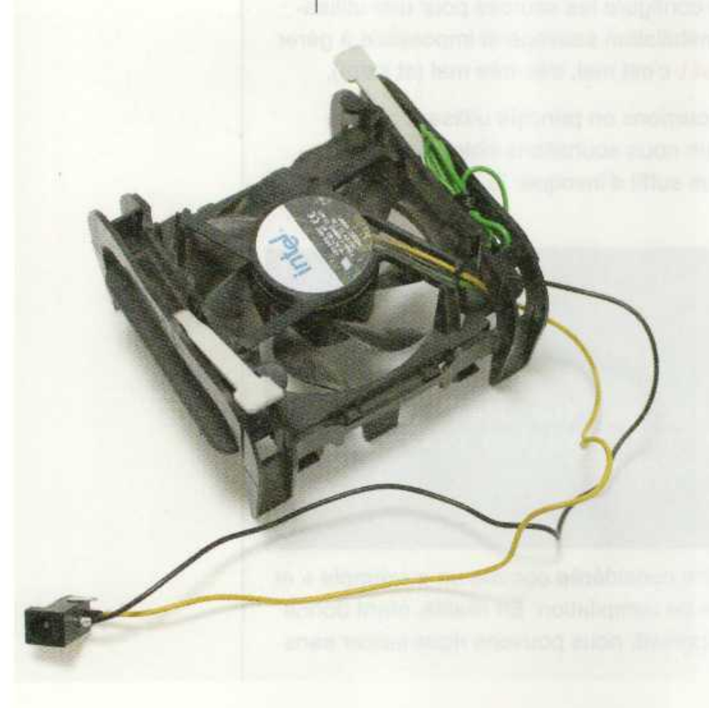
Il existe plusieurs solutions, plus ou moins élégantes, permettant de refroidir activement le SoC Broadcom d'une Raspberry Pi. N'ayant rien sous la main de dédié à la Pi, j'ai tout simplement recyclé le ventilateur d'un Ventirad Intel, légèrement modifié pour être alimenté avec un bloc secteur 12V. Pas très beau, mais terriblement efficace !

Comme vous pouvez le voir, tout ceci aura pris une heure et 36 minutes, mais, au final, on obtient dans `builds/xtensa-esp32-elf` une version équivalente à ce qu'Espressif propose au téléchargement pour GNU/Linux x86 (32 ou 64 bits).

On pourra laisser le répertoire dans l'état pour utiliser la chaîne de compilation, après avoir rendu l'arborescence inscriptible (comme un répertoire standard) avec :

```
$ chmod -R u+w builds/xtensa-esp32-elf
```

Dès lors il sera possible d'utiliser la chaîne avec le chemin `~/ESP32/crosstool-NG/builds/xtensa-esp32-elf` ou copier tout le répertoire `xtensa-esp32-elf` et son contenu à l'endroit qui vous conviendra davantage (`/opt/ESP23` ?).



2. D'ABORD, INSTALLER LE STRICT MINIMUM

Les explications qui vont suivre partent du principe qu'elles seront misent en œuvre avec un système GNU/Linux et plus précisément une distribution Debian/Ubuntu. L'utilisation d'une autre distribution est bien entendu possible à condition d'adapter une petite partie des commandes. Il devrait également être possible d'appliquer cette procédure dans un environnement WSL de Windows 10, mais ceci n'a pas été testé. Enfin, ceci pourrait également être applicable sur une Raspberry Pi à condition de disposer d'une chaîne de

compilation appropriée. Celle-ci est, par défaut, proposée par Espressif uniquement pour les architectures PC x86 32 et 64 bits et il sera donc nécessaire de compiler soi-même la chaîne de compilation, opération fastidieuse et longue.

Avant toute chose nous devons planter le décor et équiper le système des outils indispensables pour la suite des opérations. Nous avons d'une part ce que le système peut prendre en charge sous forme de paquets et de l'autre un élément fourni via les serveurs d'Expressif : la chaîne de compilation permettant de compiler les programmes devant être exécutés par l'ESP32.

L'installation des paquets se fera simplement avec :

```
$ sudo apt-get install git wget make \
libncurses-dev flex bison gperf python python-serial
```

La chaîne de compilation devra être téléchargée et installée manuellement dans, par exemple, un sous-répertoire de votre répertoire personnel. Il existe deux versions utilisables en fonction de votre architecture : 32 ou 64 bits. Utilisez la commande `uname -m` pour connaître la vôtre. Si `i686` s'affiche, vous êtes en 32 bits, si c'est `x86_64` vous êtes en 64 bits.

La page **https://esp-idf.readthedocs.io/en/latest/get-started/linux-setup.html** tient à jour un lien de téléchargement pour chaque architecture (**https://dl.espressif.com/dl/**). À cette date, les deux fichiers sont respectivement `xtensa-esp32-elf-linux64-1.22.0-80-g6c4433a-5.2.0.tar.gz` et `xtensa-esp32-elf-linux32-1.22.0-80-g6c4433a-5.2.0.tar.gz`, mais ceci peut, bien entendu, évoluer.

Récupérez l'archive puis décompressez-la dans un répertoire dédié que vous aurez fraîchement créé (`~/ESP32` par exemple) :

```
$ cd ~
$ mkdir -p ESP32
$ tar xfzv xtensa-esp32-elf-linux32-1.22.0-80-g6c4433a-5.2.0.tar.gz -C ~/ESP32
$ cd ESP32
$ ls -F
xtensa-esp32-elf/
```

Dans ce que vous venez de désarchiver se trouve un répertoire `bin/` contenant tous les programmes utilisables. Comme tout ceci n'a pas été installé dans un chemin que le système connaît, il ne peut donc retrouver les commandes associées. Pour changer cet état de fait, il faut alors ajouter `~/ESP32/xtensa-esp32-elf/bin` dans votre chemin de recherche en éditant votre fichier `~/.bashrc` et en ajoutant une ligne :

```
export PATH="$HOME/ESP32/xtensa-esp32-elf/bin:$PATH"
```

Dès le prochain démarrage, la prochaine connexion ou la prochaine ouverture d'une fenêtre de terminal, tout le contenu sera alors accessible. Vous pouvez d'ailleurs vérifier cela très facilement en appelant le compilateur GCC avec :

```
$ xtensa-esp32-elf-gcc --version
xtensa-esp32-elf-gcc (crosstool-NG crosstool-ng-1.22.0-80-g6c4433a) 5.2.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Nous avons bien un GCC version 5.2.0 qui parait fonctionnel et pouvons maintenant passer à la suite. La chaîne de compilation permet simplement de produire des binaires pouvant être exécutés par l'ESP32, mais nous avons besoin de tout le reste (bibliothèques, entête, FreeRTOS, etc.). Nous devons installer l'IDF, le framework de développement ESP32 d'Espressif.

3. INSTALLATION DE L'ESP-IDF

Tout comme la chaîne de compilation, l'IDF peut être installé où bon vous semble, du moment que vous configurez le système de façon à ce qu'il le retrouve. Comme il n'est pas dans nos habitudes de nous disperser, nous allons l'installer non loin de la chaîne de compilation, dans un sous-répertoire de **~/EPS32**. Pour ce faire, nous allons utiliser Git pour cloner le dépôt officiel :

```
$ cd ~/ESP32
$ git clone --recursive https://github.com/espressif/esp-idf.git
Clonage dans 'esp-idf'...
remote: Counting objects: 43244, done.

[...]

Clonage dans '/home/denis/ESP32/esp-idf/components/bt/lib'...
Clonage dans '/home/denis/ESP32/esp-idf/components/coap/libcoap'...
Clonage dans '/home/denis/ESP32/esp-idf/components/esp32/lib'...
Clonage dans '/home/denis/ESP32/esp-idf/components/esptool_py/esptool'...
Clonage dans '/home/denis/ESP32/esp-idf/components/libsodium/libsodium'...

[...]

Chemin de sous-module 'components/nghttp/nghttp2/third-party/neverbleed' :
'da5c2ab419a3bb8a4cc6c37a6c7f3e4bd4b41134' extrait
Chemin de sous-module 'components/spiffs/spiffs' :
'f5e26c4e933189593a71c6b82cda381a7b21e41c' extrait
```

Notez l'utilisation de l'option **--recursive** nous permettant de récupérer par la même occasion tous les sous-modules Git. Si vous oubliez cette option, vous pouvez entrer dans le répertoire **esp-idf** nouvellement créé et utiliser **git submodule update --init**.

Là encore, pour que le système retrouve ses petits nous devons modifier notre configuration en éditant le fichier **~/.bashrc** afin de déclarer une variable d'environnement contenant le chemin où est installé l'IDF. Une simple ligne fera l'affaire :

```
export IDF_PATH="$HOME/ESP32/esp-idf"
```

Dès la prochaine utilisation de la ligne de commandes, la variable **$IDF_PATH** contiendra le nécessaire et les éléments de l'IDF pourront être trouvés par le système de compilation. Nous pouvons d'ailleurs le vérifier avec :

```
$ echo $IDF_PATH
/home/denis/ESP32/esp-idf
```

4. CRÉER UN PROJET ET UTILISER LE COMPOSANT ARDUINO

À ce stade, l'ensemble des éléments permettant de développer pour ESP32 est installé et utilisable, mais il nous manque une dernière brique permettant de le faire avec les bibliothèques et donc le « langage » propre à Arduino. Cette brique prend la forme d'un composant, un « greffon », supplémentaire devant être installé non pas dans l'environnement fraîchement installé, mais en compagnie de chaque projet que nous allons créer.

Pour tester tout cela, nous devons donc, avant toutes choses, créer un premier projet. Ici, pas de menu ou de bouton dans une interface graphique, ceci se fait en créant des répertoires et des fichiers. Là encore, pour maintenir une certaine cohérence, nous allons tout simplement ajouter cela dans notre répertoire **~/ESP32**, aux côtés de la chaîne de compilation et le l'IDF :

Développer des croquis Arduino pour ESP32 peut se faire directement dans l'environnement Arduino, après avoir installé le support via le gestionnaire de cartes. Cette solution est pratique, rapide et simple, mais n'offre pas accès à toutes les fonctionnalités des cartes ESP32 et de FreeRTOS.

```
$ cd ~/ESP32
$ mkdir -p premier/main
$ cd premier
```

Dans notre répertoire du projet, **~/ESP32/premier**, nous commençons par créer un **Makefile** contenant des lignes intelligibles par la commande **make** et référençant directement l'IDF :

```
PROJECT_NAME := premier
include $(IDF_PATH)/make/project.mk
```

La ligne **include** utilise le contenu de la variable d'environnement **IDF_PATH** pour inclure le fichier **project.mk** contenant la « recette » pour la compilation d'un programme reposant sur l'IDF.

Dans le sous-répertoire **main/**, nous ajoutons notre croquis Arduino de base (qui n'est autre qu'un simple fichier C++) :

```
#include <Arduino.h>

void setup(){
  Serial.begin(115200);
}

void loop(){
        Serial.println("loop");
        delay(1000);
}
```

Pour compléter la configuration, nous l'accompagnons d'un fichier **component.mk**, indispensable, mais ici vide :

```
% touch component.mk
% cd ..
```

Au final, nous avons donc créé l'arborescence suivante :

```
~/ESP32/
   premier/
     Makefile
     main/
       component.mk
       premier.cpp
```

Ceci est un projet dans le sens ESP-IDF du terme. Notez que nous incluons **Arduino.h** dans notre source **premier.cpp**, mais, pour l'heure, l'IDF ne supporte en aucune manière la compilation des croquis Arduino et ne saura donc pas où trouver ce fichier. Nous devons ajouter un composant permettant cela, directement dans le projet :

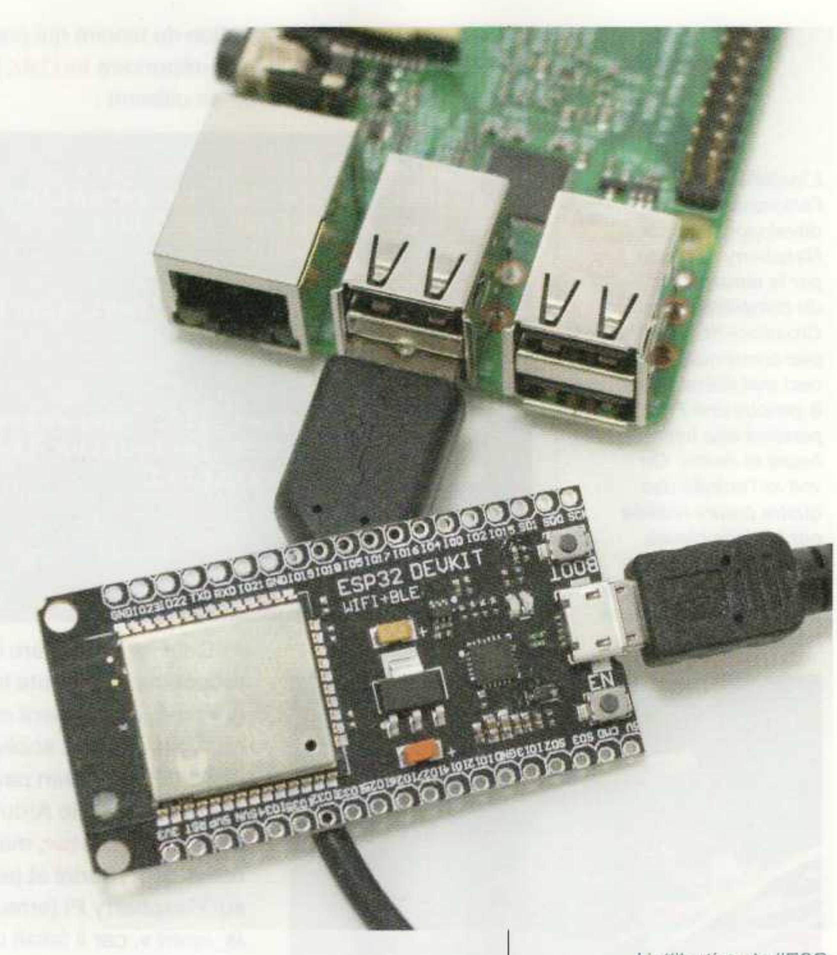
```
$ mkdir components
$ cd components
$ git clone --recursive https://github.com/espressif/arduino-esp32.git arduino
Clonage dans 'arduino'...
remote: Counting objects: 7132, done.
[...]
Clonage dans '/home/denis/ESP32/premier/components/arduino/libraries/BLE'...
Chemin de sous-module 'libraries/BLE' :
'6bad7b42a96f0aa493323ef4821a8efb0e8815f2' extrait
```

Tout est prêt, mais nous devons encore configurer le tout et en particulier le composant Arduino. Pour ce faire, nous utilisons la commande **make menuconfig** à la racine du projet (**~/ESP32/premier**). Dans l'interface à menus qui se présente à nous, nous devons faire un tour dans *Component config* puis *Arduino Configuration*. Plusieurs options sont activables :

- *Autostart Arduino setup and loop on boot* : active le fonctionnement classique Arduino avec les fonctions **setup()** et **loop()** (dans le cas contraire, nous devons écrire une fonction **main()** ;

- *Autoconnect WiFi on boot* (apparaît avec l'activation de la précédente option) : connexion automatique au Wifi dernièrement configuré ou utilisation via la méthode **WiFi.begin()** ;
- *Disable mutex locks for HAL* : option pour le multitâche permettant de désactiver l'accès exclusif (mutex) au matériel (HAL pour *Hardware Abstraction Layer*).

Vous devrez également faire un tour dans *Compiler options*, à la racine du menu, et activer l'option *Enable C++ exceptions*, nécessaire à la compilation de croquis Arduino. Ceci est susceptible d'être activé par défaut dans une future version du composant (c'était le cas avec la version précédente). Vous pourrez également visiter le menu *Serial flasher config* afin de configurer différents points concernant le matériel, dont le port série utilisé pour la programmation, la vitesse, la taille de la flash, etc. Ceci correspond à la configuration de la plateforme dans le menu *Outils* de l'IDE Arduino. La taille de la flash sur un module WROOM-32 est généralement de 4 Mo (32 Mbits), en mode DIO, à 40 Mhz.



L'utilisation de l'ESP-IDF vous permettra de développer du code en C/C++ ou en « langage Arduino » directement sur une Pi. Ceci nécessite cependant un certain nombre de manipulations techniques relativement longues, mais il n'existe, pour l'instant, pas d'autres solutions.

Choisissez vos options et quittez l'interface de configuration en sauvegardant au moment de l'apparition du message « *Do you wish to save your new configuration?* ». Cette configuration sera enregistrée dans le fichier **sdkconfig** et vous pourrez appeler **make menuconfig** pour éventuellement changer ces options par la suite, selon vos préférences ou les résultats de vos expérimentations. Il ne vous reste plus ensuite qu'à lancer la compilation avec :

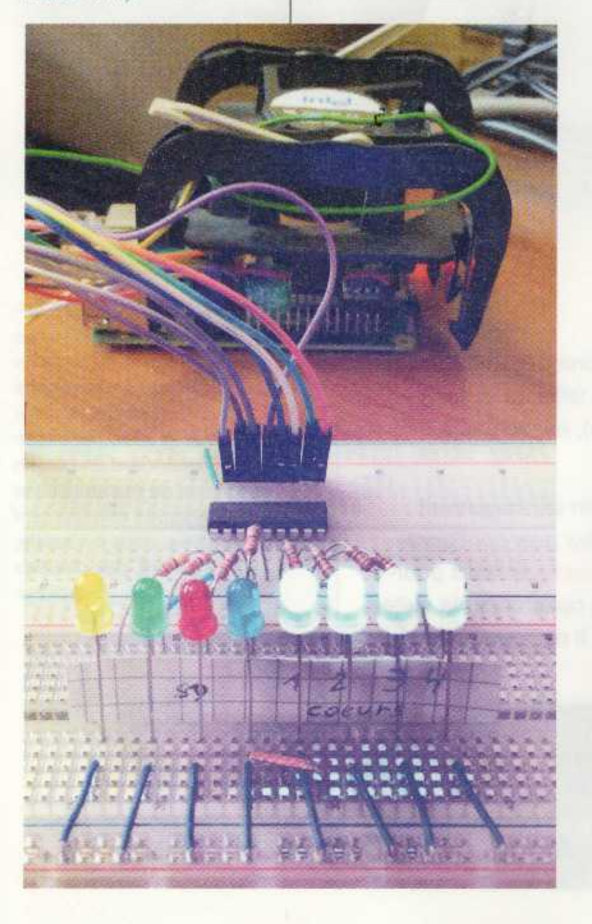
```
$ make
[...]
CC build/xtensa-debug-module/trax.o
AR build/xtensa-debug-module/libxtensa-debug-module.a
LD build/premier.elf
```

L'ensemble de la construction du binaire qui prendra place dans la flash du module ESP32 se trouve dans le sous-répertoire **build/**. Une fois votre carte connectée en USB, vous pourrez la programmer en utilisant :

```
$ make flash
Flashing binaries to serial port /dev/ttyUSB1
  (app at offset 0x10000)...
esptool.py v2.1
Connecting........_
Chip is ESP32D0WDQ6 (revision 0)
Uploading stub...
Running stub...
[...]
Wrote 3072 bytes (122 compressed) at 0x00008000
  in 0.0 seconds (effective 1643.1 kbit/s)...
Hash of data verified.


Leaving...
Hard resetting...
```

L'installation de l'environnement de développement sur Raspberry Pi passe par la construction du compilateur avec Crosstool-NG. Ce n'est pas compliqué, mais ceci met littéralement à genoux une Pi 3 pendant une bonne heure et demie. On voit ici l'activité des quatre cœurs notifiée par les leds bleues durant l'opération (configuration que nous avions détaillée dans le numéro 17).



Comme la dernière ligne le précise, le module a été redémarré et exécute immédiatement le nouveau programme, exactement comme avec l'environnement Arduino. Vous pourrez accéder au port série avec l'outil de votre choix (GNU Screen par exemple) pour obtenir un équivalent du moniteur série Arduino. Il est également possible d'utiliser **make monitor**, mais celui-ci demande une version récente de PySerial et peut donc poser quelques problèmes sur Raspberry Pi (erreur « *Serial object has no attribute is_open* », car il fallait utiliser la méthode **isOpen()** dans les anciennes versions et non l'attribut **is_open**). Bravo, vous avez écrit, compilé, enregistré et exécuté votre premier croquis Arduino avec l'IDF Espressif.

POUR FINIR : QU'AVONS-NOUS OBTENU ?

L'environnement Arduino n'est pas un modèle d'ergonomie et de richesse de fonctionnalités, mais il est simple d'utilisation et accessible. Inversement, l'utilisation de l'IDF d'Espressif en compagnie du composant Arduino est plus spartiate dès lors qu'on n'a pas l'habitude d'utiliser des outils très teintés UNIX (**make**, ligne de commandes, éditeur de code, Git, etc.), mais il donne également accès à bien plus de fonctionnalités.

Il est ainsi possible d'activer des fonctions avancées de FreeRTOS, voire de supprimer certains éléments pour

optimiser son code. Plus technique, il devient également possible de déboguer ses programmes avec des outils avancés comme OpenOCD et GDB et de contrôler leur exécution avec une sonde JTAG. Ceci permet de répondre à des besoins qui sortent généralement du cadre de la simple écriture de croquis Arduino, mais qu'il est intéressant d'avoir sous la main ponctuellement pour des projets spécifiques.

À cela s'ajoute le fait que de plus en plus d'utilisateurs réalisent leurs projets avec l'ESP-IDF et non avec l'environnement Arduino. Dès lors qu'on cherche des informations précises sur des réalisations à base d'ESP32, beaucoup de résultats mentionnent l'installation de l'IDF comme prérequis et font totalement l'impasse sur la partie Arduino. Disposer d'une installation fonctionnelle est donc une bonne idée même si l'on ne souhaite pas forcément en faire un usage systématique.

L'utilisation de l'IDF et de ce mode de développement particulier donne également l'occasion de mettre son nez dans le fonctionnement interne de FreeRTOS et d'explorer le domaine. Sans oublier, bien sûr, que tout ceci est ouvert et que vous pouvez, si le cœur vous en dit, participer aux développements en proposant des contributions et des corrections de bugs via GitHub. À l'heure où cet article est rédigé, il est encore relativement difficile d'intégrer rapidement une bibliothèque Arduino dans un projet ESP-IDF/Arduino (puisque la gestion des bibliothèques est l'affaire de l'environnement Arduino et non du compilateur). Ceci pourrait être la pierre que vous apporterez à l'édifice...

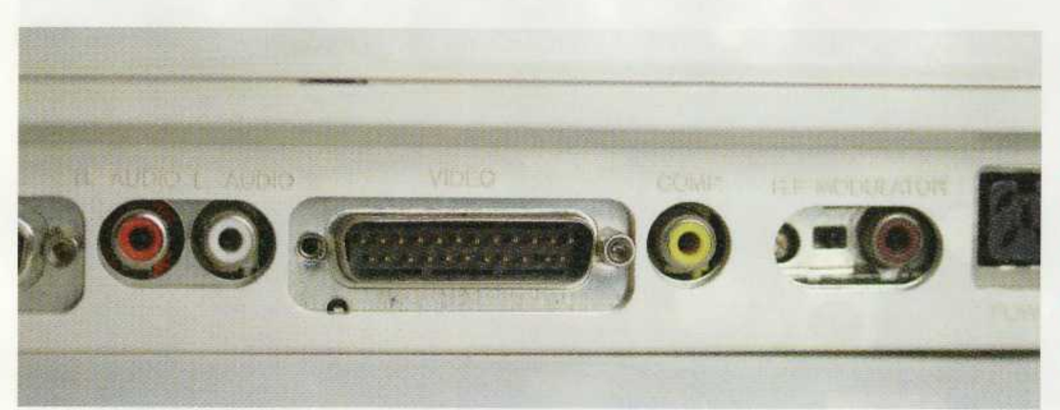
La décision d'investir votre temps et votre énergie est vôtre, mais le simple fait que l'opportunité de programmer autrement soit disponible ne peut être que célébré. À vous maintenant de choisir la marmite qui vous convient pour cuisiner votre ESP32... DB

SIGNAUX VIDÉOS ANALOGIQUES : COMMENT VOUS Y RETROUVER ?

Denis Bodor



Composite, S-Video, RGB, Péritel, RF, VGA... Tous ces noms désignent des « espèces » en voie de disparition. Non des espèces animales ou végétales, mais celles de standards vidéos qui étaient d'usage il y a peu de temps encore, mais à présent exterminées une à une par quelques normes, totalement numériques, parmi lesquelles l'universel HDMI. En quoi est-ce un problème ? La réponse tient en quelques mots : anciennes consoles & anciens ordinateurs...



Un ordinateur comme cet Amiga 1200 propose une large gamme de sorties vidéos. De gauche à droite, après les sorties audios, on trouve ainsi le connecteur D-sub 23 proposant RGB, la sortie composite et celle du modulateur RF.

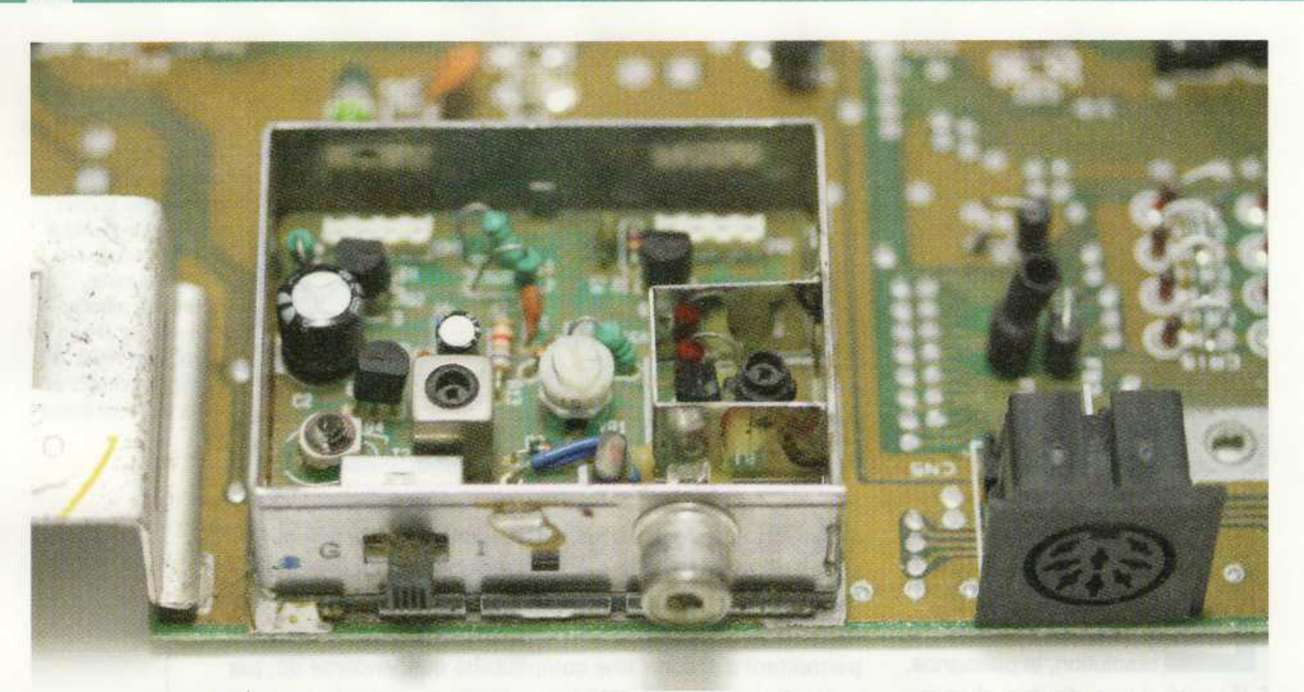
Le progrès technologique est formidable, plus le temps passe plus la vitesse, la qualité, la résolution, la puissance, l'efficacité et bien d'autres choses, augmentent dans les technologies qui nous entourent (ou devrait-on dire « nous encerclent » ?). C'est ainsi que les moniteurs et téléviseurs à tube cathodique se sont vu remplacer par les écrans LCD, une technologie « matricielle » très différente qui a, par la suite, justifié d'autres évolutions comme l'augmentation de la résolution puis, finalement, l'abandon des signaux analogiques pour transmettre les images devant s'afficher à l'écran.

Aujourd'hui, cette évolution en est arrivée à un stade où une entrée analogique commence à devenir une denrée rare sur un téléviseur et a presque totalement disparu sur un écran de PC flambant neuf. On trouve donc des connecteurs HDMI, DVI, DisplayPort et/ou Thunderbolt sous différentes formes et respectant différentes versions des normes et standards, mais le VGA a presque totalement disparu des moniteurs PC et les ports composite, S-vidéo et péritel commencent à être oublié des constructeurs.

Ceci est cependant parfaitement logique puisqu'une certaine dynamique existe entre les équipements. À quoi bon conserver un connecteur et la gestion du signal pour lequel il est initialement prévu si plus aucun équipement n'est encore fabriqué pour s'y brancher ? Une phase transitoire existe, permettant d'assurer une compatibilité descendante où, par exemple, un nouveau téléviseur devra supporter la connexion d'un lecteur DVD ou d'un caméscope déjà en possession de l'utilisation, mais n'offrant qu'une sortie analogique. Cette phase est celle dans laquelle nous sommes actuellement dans le domaine de la connectique vidéo.

Mais ne vous y trompez pas, il s'agit d'une compatibilité « forcée », ce que fait un téléviseur se résume à une simple conversion analogique vers numérique avec, en guise de base de critère de qualité, le traitement d'un certain type d'images qu'on pourrait qualifier de « naturelles » (vidéo amateur, film, séries, documentaires, etc.). Absolument aucun constructeur de téléviseurs ne prend en compte avec intérêt le traitement et la conversion d'un signal vidéo transportant des images présentant de forts contrastes et des formes aux bordures nettes. Autrement dit, le traitement des signaux analogiques provenant de consoles et d'ordinateurs ne figure absolument pas dans leur cahier des charges. C'est triste mais, bien souvent, brancher ce genre de matériel sur un écran à tube cathodique donnera une bien meilleure image que sur une TV dernier cri.

Certains diront que ceci est bien normal et que la seule authentique façon d'utiliser des consoles de jeu et ordinateurs familiaux vintages consiste à les brancher sur des TV et moniteurs de leur époque, afin de vivre les vraies sensations et une véritable expérience « années 80 ». À chacun de tracer la ligne où bon lui chante, mais personnellement ce n'est pas ce que je cherche (sinon, pour l'occasion, il faudrait aussi adopter la coupe de cheveux et les vêtements de cette période et là, subitement c'est moins amusant). C'est une question d'interprétation, mais moi je préfère voir cela comme « redécouvrir un jeu tel qu'il aurait dû être » plutôt que comme « le programmeur avait prévu qu'il s'affiche avec les moyens de l'époque ».



Un modulateur RF, comme celui du Commodore 64, est chargé d'utiliser le signal vidéo produit par la machine et d'en faire presque littéralement une chaîne de télévision. Après connexion, on réglait alors qu'un des boutons de sélection des chaînes sur la fréquence correspondante (canal) et on obtenait image et son comme avec une réception hertzienne normale.

Quoi qu'il en soit, avant l'arrivée du « tout numérique », il existait bien des moyens pour qu'une machine produise et transmette une image. À l'époque, la démarche était relativement simple, on branchait tout bonnement le matériel à la prise dont on disposait. Aujourd'hui, ces différences entre normes et standards, ont une importance bien plus grande car, que l'on convertisse le signal avec un matériel spécialisé ou en laissant faire le téléviseur, la qualité du résultat est directement proportionnelle à celle du signal de départ. Et tous les signaux ne se valent pas...

1. ANATOMIE D'UN SIGNAL VIDÉO

Pour afficher une image sur un écran reposant sur le principe utilisé par les tubes cathodiques, un certain nombre de signaux doivent exister. Le principe de base consiste en un balayage de l'écran d'en haut à gauche à en bas à droite, ligne par ligne. Peu importe que le signal transporte une image en couleur ou non, le principe est le même, mais il est plus simple à imaginer avec une image monochrome et sans nuances.

Dans ce cas hypothétique, nous avons besoin de trois signaux pour piloter le balayage. Le premier signal détermine ou cadence le retour du faisceau en haut à gauche de l'écran pour démarrer l'affichage d'une nouvelle image (ou trame). Ce signal est celui de la **synchronisation verticale** ou **V-SYNC** et sa fréquence détermine le nombre d'images par seconde qui seront affichées, qui peut être égal à cette valeur ou la moitié (cf. un peu plus bas).

Un signal similaire existe, permettant de cadencer le retour du faisceau à gauche de l'écran pour commencer une nouvelle ligne, c'est le signal de **synchronisation horizontale** ou H-SYNC. D'où la notion de balayage.

Enfin, nous avons le signal des données vidéos en elles-mêmes qui arrivent brutes et donc réparties ligne par ligne, écran par écran, grâce à V-SYNC et H-SYNC. On peut aisément imaginer le processus en l'illustrant ainsi : vous avez deux métronomes distincts, un qui fait « toc », un qui fait « ding » et vous tracez une ligne au feutre sur une feuille de papier en faisant varier la pression du tracé. À chaque fois que vous entendez un « toc » du métronome, vous levez votre feutre et commencez une nouvelle ligne. À chaque fois que c'est un « ding » vous changez de feuille et revenez en haut à gauche pour un nouveau dessin. Le « toc » est H-SYNC, le « ding » est V-SYNC

et la pression que vous appliquez sur le feutre constitue les données du dessin.

Pour continuer dans cette analogie, il faut distinguer deux façons de remplir une feuille. Nous avons tout d'abord la méthode progressive et la plus simple, correspondant à celle décrite à l'instant. On dessine progressivement le tracé de haut en bas, ligne pas ligne. Le problème avec cette technique est la limite en termes de vitesse, et pour obtenir 20 feuilles par seconde pour une animation fluide vous devez tout dessiner relativement rapidement. Si vous ne le faites pas, le rafraîchissement sera visible et l'animation saccadée.

Pour économiser du travail tout en maintenant la fluidité de l'animation, il existe une technique très simple consistant à tracer une ligne sur deux. Plutôt que de tracer de façon progressive, on commence par tracer toutes les lignes impaires puis toutes les lignes paires. Chaque feuille aura donc la moitié des informations d'une image mais, si elles sont changées suffisamment rapidement, la tricherie ne sera pas perceptible. Techniquement, il y a cependant deux fois moins de lignes à tracer avec la même fréquence de synchronisation (on ne touche pas aux métronomes) et on obtient deux fois plus d'images par seconde. Cette technique est appelée l'entrelacement vidéo ou *interlaced video* en anglais.

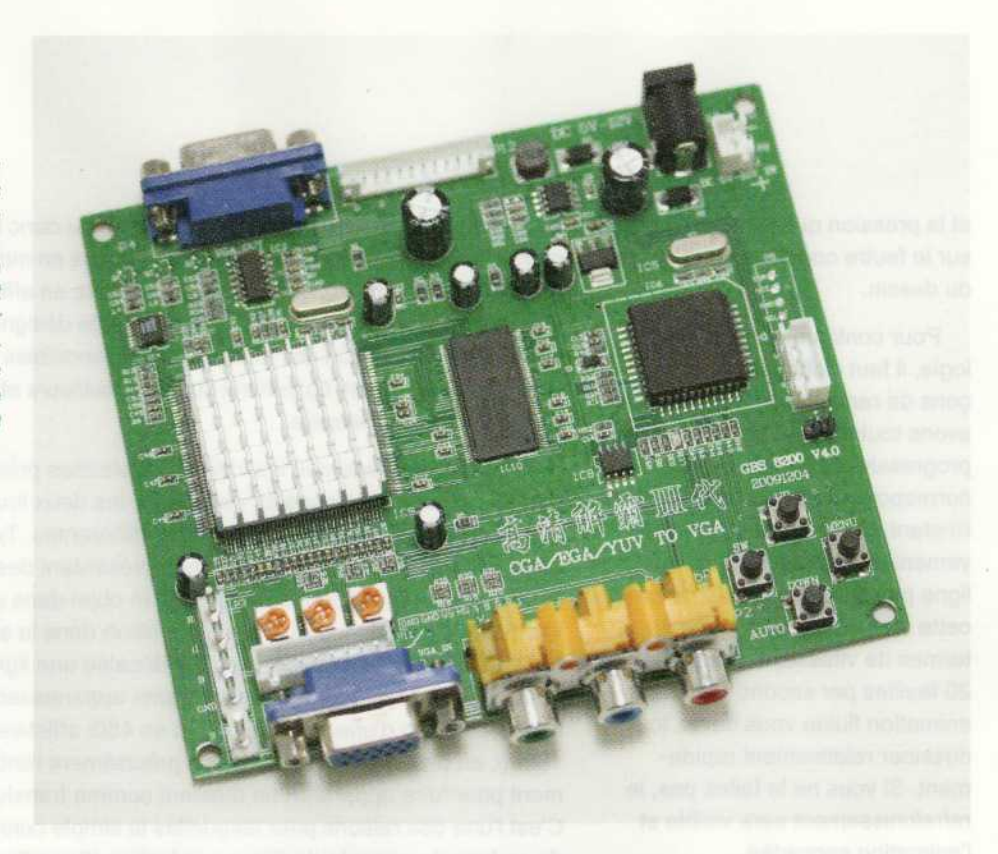
Ces deux modes existent toujours aujourd'hui et sont la raison d'être des lettres « p » pour *progressive* et « i » pour « interlaced » en suffixe des résolutions vidéos. 720p désigne donc une image de 720 pixels de large avec un affichage en mode progressif et 1080i une image de 1080 pixels avec un affichage entrelacé. On retrouve par exemple ce type de désignation dans la documentation des modes vidéos disponibles pour la Raspberry Pi, mais également sur les téléviseurs et autres équipements audiovisuels.

Notez au passage que les vidéos entrelacées présentent parfois des artefacts appelés *combing* si les deux images utilisées pour l'entrelacement sont très différentes. Typiquement, cela se produit avec des vidéos présentant des objets se déplaçant rapidement à l'écran, où un objet dans une image n'est plus du tout à la même position dans la suivante. L'image une fois recombinée semble décalée une ligne sur deux. Ceci est typiquement un problème apparaissant avec la source vidéo d'une console de jeu en 480i affichée en 1080p, en particulier si le jeu utilise précisément l'entrelacement pour faire apparaître un élément comme translucide. C'est l'une des raisons pour lesquelles la simple connexion de ce type de console de 6ème génération (Sega Dreamcast, Nintendo GameCube, Sony PS2, etc.) sur un téléviseur donne généralement des résultats assez pauvres, la TV n'étant pas conçue pour traiter (désentrelacer) proprement ce type de contenu et les images présentant des éléments se déplaçant rapidement.

Idéalement, il faut donc préférer un matériel dédié pour la conversion, comme par exemple un XRGB-Mini Framemeister ou un OSSC, qui non seulement procédera à la conversion propre du signal analogique, mais augmentera la résolution tout en désentrelaçant convenablement les images. Ce type de produits est généralement désigné via le terme d'*upscaler* puisqu'il fournit au moniteur ou à la TV une vidéo dans un format « augmenté » directement et nativement utilisable.

2. PETIT BESTIAIRE DES SIGNAUX ANALOGIQUES

Que vous regardiez à l'arrière d'un Commodore 64, d'une SNES ou d'un Atari ST, vous trouverez toujours un ou plusieurs connecteurs permettant de brancher un moniteur ou une TV. N'espérez pas cependant y voir la moindre trace de sortie numérique type DVI ou HDMI. L'idée même de pouvoir, avec une machine de cette époque, envoyer une image sous la forme d'une série d'octets était tout bonnement impensable.



Pour brancher une source à un moniteur, il n'est pas forcément nécessaire d'opter pour du numérique. Avec ce type de carte économique initialement prévue pour mettre à jour une borne d'arcade (GBS-8200), il devient possible relativement facilement de convertir un signal YUV ou RGB vers VGA.

Les signaux vidéos étaient donc générés analogiquement par la puce graphique du matériel de façon à permettre la transmission d'une quantité importante d'informations à destination d'un système d'affichage qui ne procédait absolument pas à la conversion inverse.

Un écran à tube cathodique ne possède, en effet, aucun élément numérique et peut être vu comme une sorte de super-oscilloscope. Les signaux analogiques sont directement utilisés pour contrôler le balayage d'un faisceau d'électrons qui, en frappant la surface interne de l'écran, produit de la lumière. En simplifiant grossièrement les choses donc, c'est directement le signal ou les signaux envoyés par l'émetteur qui pilotent le mouvement du faisceau et dessinent l'image.

La manière dont ces signaux sont combinés, envoyés, séparés constitue le standard ou la norme utilisée, et il en existe plusieurs.

2.1 RF / antenne

Sans le moindre doute possible ceci est la pire option possible. L'idée repose sur une approche toute simple : comme les consoles et ordinateurs devaient pouvoir être connectés à un téléviseur, et que les téléviseurs sont à la base conçus pour recevoir une émission « hertzienne », le signal se présentait exactement comme une telle émission.

En lieu et place de l'antenne du téléviseur prenait donc place un câble connecté à un modulateur intégré dans la machine. Celui-ci avait pour tâche de prendre les signaux vidéos et audios et les moduler sur une fréquence correspondant à celle d'une chaîne TV (canal). L'ordinateur familial agissait donc comme un émetteur TV et il suffisait de régler le bon canal sur le téléviseur pour voir l'image apparaître.

L'ensemble des signaux (couleurs, synchronisation, etc.) étant combinés et modulés, le moindre bruit parasite dégradait donc l'image. Pire encore, très souvent la modulation elle-même induisait des interférences non seulement dans le signal vidéo, mais également dans les composants se trouvant autour du modulateur, généralement blindé en étant placé dans un boîtier métallique.

Aujourd'hui, les téléviseurs n'étant plus compatibles avec les

émissions analogiques et les moniteurs n'ayant jamais eu de tuner intégré, la question de l'utilisation de ce signal ne se pose plus. Sauf, bien sûr, si l'on cherche à expérimenter dans ce domaine avec un vieux téléviseur. Pour quelques ordinateurs, donc le ZX Spectrum de Sinclair, il est même généralement recommandé de modifier, désactiver ou supprimer le modulateur HF pour améliorer la qualité du signal obtenu via un autre connecteur ou à un autre endroit sur le circuit imprimé.

2.2 Composite

Voici sans le moindre doute le connecteur le plus présent et le plus utilisé, mais certainement pas celui le plus intéressant. Un signal composite est accessible via un connecteur de type cinch (RCA) jaune. C'est l'un des connecteurs typiques que l'on trouve accompagné d'un blanc et d'un rouge pour les signaux audios respectivement gauche et droite (en stéréo).

Contrairement au signal du modulateur RF, celui-ci ne transporte que l'image, mais regroupe en un seul conducteur les informations sur la couleur, la luminosité, la synchronisation verticale et horizontale. Tout ce petit monde se retrouve combiné et transporté sur un unique câble, d'où le nom de vidéo composite.

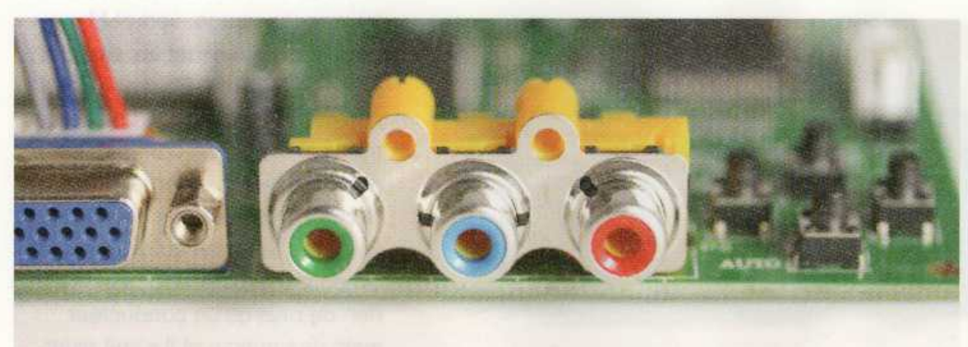
L'information de couleur est encodée selon une norme qui peut être PAL, NTSC ou SECAM. Dans les grandes lignes, ceci explique pourquoi, dans votre jeunesse, une console ou un ordinateur avec une sortie composite PAL ne pouvait afficher qu'une image en noir et blanc sur un téléviseur uniquement SECAM (et non PAL/SECAM). Et ce alors même que l'image elle-même était parfaitement visible.

Ce mélange de signaux découle sur une image baveuse et parasitée qui était relativement acceptable avec un écran à tube cathodique du fait de procéder l'affichage lui-même. Avec un écran LCD en revanche, et donc des pixels bien définis, le bruit et les débordements de couleurs s'en trouvent amplifiés ou du moins rendus bien plus perceptibles. L'utilisation, avec un ordinateur ou une console des années 80, d'une entrée composite sur un téléviseur moderne vous donne généralement l'impression que l'image était bien meilleure dans vos souvenirs, et pour cause, c'était le cas.

Même en utilisant un upscaler spécialement prévu pour traiter le signal composite provenant d'un ordinateur ou d'une console vintage, et malgré toutes les fonctionnalités destinées à proprement retravailler l'image, le résultat ne sera certainement pas à la hauteur de vos attentes. La sortie composite sera donc le choix universel, certes, mais aussi celui fait par dépit s'il n'existe pas d'autres solutions pouvant être mises en œuvre (plus ou moins facilement).

2.3 S-Video

Vous l'avez compris, l'intégrité des signaux vidéos est proportionnelle au nombre de conducteurs utilisés. Dans ce sens, la meilleure option après la sortie composite sera S-Video.



Ce connecteur composé de trois prises RCA, sur le GBS-8200, n'est pas destiné à recevoir un signal RGB, mais YUV (YPbPr) et ce malgré le fait que les connecteurs soient effectivement rouge, vert et bleu. L'entrée RGB, elle, se présente sous forme de broche, ailleurs sur le circuit.

Ce standard également appelé Y/C (parfois aussi S-VHS, qui est un abus de langage) ne transporte pas non plus de signal audio mais, de plus, l'image est transmise via deux signaux distincts, la luminance (signal transmettant l'intensité lumineuse de la vidéo) et la chrominance (signal véhiculant l'information de couleur). Les deux signaux sont identifiés respectivement par les lettres « Y » et « C ». L'information de chrominance est encodée selon un standard précis décliné en version PAL ou NTSC (plus rare), il n'y a pas de standard SECAM en S-Video.

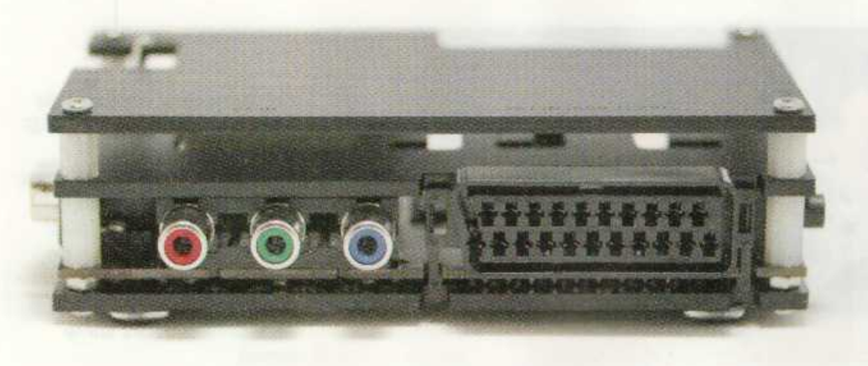
Les signaux de synchronisation horizontal et vertical sont véhiculés avec celui de la luminance. Il est possible de très facilement convertir un signal S-Video en composite en connectant les deux conducteurs et en plaçant, en parallèle, un condensateur de quelques centaines de picofarads. Bien entendu, ceci ne devra être envisagé que dans des situations bien spécifiques puisque le signal S-Video est généralement de meilleure qualité.

Le connecteur S-Video se présente sous la forme d'un mini-DIN à quatre broches : deux pour les masses, un pour la luminance et un pour la chrominance. Il existe également des mini-DIN 7 et 9 broches non standards utilisées dans certains équipements, ajoutant des signaux vidéos analogiques supplémentaires (et redondants) comme composites, RGB ou YPbPr, mais également numériques comme un bus i2c.

Certaines machines, dont le Commodore 64, ne proposeront pas mieux que S-Video du fait du fonctionnement même de la puce graphique intégrée et produisant uniquement ces deux signaux. Si convertir ce signal en composite est relativement simple, séparer les signaux plus avant nécessite un matériel spécifique. C'est un problème qu'on rencontre par exemple avec le C64 et un upscaler OSSC ne proposant pas d'entrée S-Video, et qui tantôt s'avère très difficile à régler.



L'OSSC pour « Open Source Scan Converter » est un projet open source et open hardware créé dans le but de proposer un upscaler totalement ouvert capable d'utiliser un signal vidéo analogique et de produire une sortie numérique HDMI. Il supporte les entrées RGB (sync-on-composite, CSYNC et sync-on-green), YPbPr et VGA (RGBHV pour RGB + H-SYNC+ V-SYNC), mais ni composite, ni Y/C (S-Video).



2.4 YUV

YPbPr ou YUV (ou encore tantôt « composantes », ou « *component* ») est une option encore plus intéressante en termes de qualité d'image. Se présentant généralement sous la forme de connecteurs cinch/RCA vert, bleu et rouge, il est important de ne pas commettre l'erreur consistant à penser qu'il s'agit là des trois couleurs fondamentales d'une synthèse additive.

« Y » transporte le signal de luminance (attention, ce n'est pas la luminance relative « Y » du S-Video) ainsi que les signaux de synchronisation vertical (trame) et horizontal (lignes). « Pb » (ou U) véhicule l'information concernant la différence entre le bleu et la luminance. Et « Pr » (ou V) transporte la différence entre le rouge et la luminance. Nous avons donc trois signaux, un pour la luminance et deux pour la chrominance.

La couleur des câbles est donc trompeuse, rouge, vert et bleu sont respectivement les signaux « Pr », « Y », et « Pb » et n'ont pas de rapport avec les couleurs transportées. J'en profiterai ici pour préciser, bien entendu, que la couleur des câbles n'a rien à voir avec le signal qui y circule. Un connecteur RCA ou cinch n'est rien de plus qu'un conducteur avec une masse et il s'agit exac-

tement du même type que ceux utilisés pour la vidéo composite (jaune) plus les deux canaux audios (rouge et blanc). Absolument rien ne vous empêche d'utiliser un tel câble pour connecter deux matériels en YPbPr, du moment que vous ne vous mélangez pas les pinceaux de bout en bout. Tantôt, sur du matériel professionnel, ces connecteurs sont remplacés par du BNC (comme pour les vieux réseaux 10base-2).

Notez qu'on parle également de « connecteur composante » pour désigner YPbPr (ce qui rajoute à la confusion), car ces trois signaux sont les composantes séparées d'une transmission vidéo. Là encore, il faut faire attention, « composante » n'est pas « composite », et cela se complique encore ensuite avec...

2.5 RGB / RVB

RGB est une séparation des signaux des trois composantes de couleur : rouge, vert et bleu. Là, les choses peuvent devenir un peu plus difficiles, car même si

Ce point est important, car il

on parle alors de « sync sur composite » (*Sync-on-composite*). Ceci ne doit pas être confondu avec la désignation « Composite Sync » (ou CSYNC) qui n'a rien à voir avec le signal ou connecteur « composite », mais signifie simplement que les signaux de synchronisation vertical et horizontal se trouvent sur leur propre connecteur. C'est donc un câble séparé qui transporte un signal composé des deux synchronisations. Ce type de câble est parfois appelé CSYNC, *Raw Sync*, *Pure Sync*, etc.

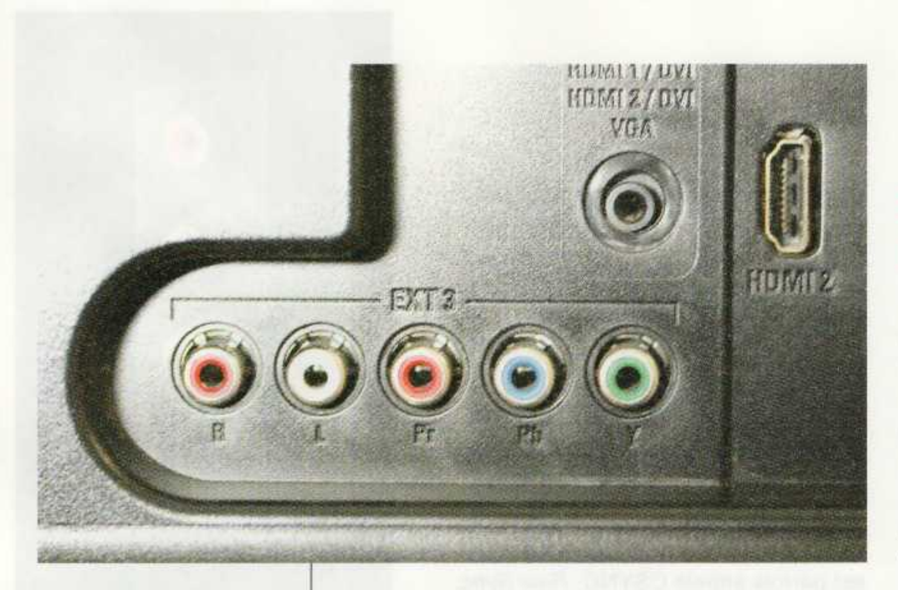
Ce n'est pas tout. Dès lors qu'on parle de vidéo, ceci ne se limite pas aux téléviseurs, mais concerne également les moniteurs informatiques plus modernes. Là, les signaux vidéos analogiques (pré DVI, HDMI et DisplayPort) sont également véhiculés par des câbles avec, généralement une séparation par couleur (RGB donc), mais des signaux de synchronisation séparés (*Separate Sync*) : un pour la synchronisation verticale (V-SYNC) et un autre pour l'horizontale (H-SYNC). C'est le cas, par exemple pour le signal VGA avec une image transportée par les signaux R, G, B, V-SYNC et H-SYNC (voir l'article dans le numéro 18 sur l'affichage VGA avec une carte Arduino et quelques résistances).

Dernier point concernant les signaux RGB et les connecteurs informatiques, le ou les signaux

Vous avez mal à la tête ? Moi aussi. En résumé, RGB/RVB n'est pas un standard, mais simplement



Parmi les connexions généralement disponibles sur un téléviseur moderne, on retrouve généralement les entrées suivantes, avec de haut en bas : audio droite, audio gauche, composite et S-Vidéo. Les consoles de jeu de 3ème à 6ème génération, comme les NES, PS2, N64, Mega Drive ou N64, était souvent livrées avec un câble audio + composite, mais ne vous y trompez pas, c'est



Quelques rares téléviseurs proposent une entrée YUV (ou YPbPr), comme ce modèle Full HD Philips 32PFL504H/12, disposant également de deux entrées HDMI, deux péritel, une composite, une VGA et une S-Video. Ceci peut sembler très intéressant, mais la conversion ainsi que l'augmentation de résolution (upscaling) intégrés ne sont souvent pas à la hauteur des attentes des amateurs de rétrogaming.

conducteur. Et enfin, parfois nous avons du RGB Sync-on-green où la synchronisation est mélangée avec le signal du vert.

Malgré toute cette complexité, l'utilisation d'une séparation RVB/RGB est le saint Graal de la transmission vidéo analogique où l'information est transmise avec un nombre maximum de conducteurs et donc une qualité optimale. Ceci explique pourquoi le VGA a été utilisé si longtemps sur PC et ce avec des résolutions pouvant aller jusqu'à 1280×1024 (voire plus parfois) tout en conservant une qualité d'affichage tout à fait acceptable pour l'époque (mais presque risible aujourd'hui).

Obtenir ces signaux d'une machine n'est pas toujours possible et lorsque c'est le cas, ce n'est pas toujours facile. Un C64, par exemple, ne vous permettra pas d'avoir accès à ces signaux, mais uniquement, au mieux, à S-Video. D'autres matériels proposent directement une sortie RGB ou encore tolèrent une modification permettant d'obtenir ces signaux. Ce genre de hacks (ou « mods ») seront plus ou moins difficiles à réaliser et il existe tantôt des modules ou des kits permettant de simplifier une bonne partie des opérations.

Ce type de modification sort du cadre du présent article, mais il suffit généralement de rechercher sur le Web le nom de la console ou de l'ordinateur concerné suivi de « RGB mod » pour trouver des informations intéressantes ou directement des modifications faites par d'autres utilisateurs.

3. LE CAS DE LA PÉRITEL

Les amateurs de rétrogaming et de rétrocomputing américains nous envient quelque chose que nous, en Europe, prenons pour acquis depuis des années : la prise péritel ou SCART pour « Syndicat des Constructeurs d'Appareils Radiorécepteurs et Téléviseurs ». Ce connecteur équipant bon nombre d'appareils audiovisuels jusqu'à très récemment est une véritable merveille.

Cette connectique normalement appelée péritélévision a été créée dans le but de simplifier l'interconnexion des équipements exploitant des signaux analogiques vidéos et audios : TV, magnétoscope, lecteur DVD, démodulateur satellite, consoles, etc. Standardisée et rendue publique dès 1978, la prise péritel s'est rapidement popularisée en Europe en étant totalement ignorée outre-Atlantique. Celle-ci est même rendue légalement obligatoire sur les TV commercialisées en France métropolitaine à partir de 1980. Ce n'est que mi-2015 que cette obligation a été levée.

La volonté de simplification qui est à la source de la création de la péritel explique à la fois son aspect physique et sa popularité. Celle-ci se présente sous la forme d'un connecteur rectangulaire muni d'un détrompeur, équipé de 21 broches. Dans cette myriade de connexions se trouvent ou peuvent se trouver les signaux :

- audio gauche et droite,
- composite,
- S-Vidéo (Y/C),
- RVG/RGB,
- YUV/YPbPr (ajouté après la création de la norme originale).

Bien entendu, un équipement n'utilisera sans doute pas tous ces signaux puisque seul un certain nombre sont décrits comme obligatoires dans la norme (composite, audio, commutations) et certaines broches peuvent avoir plusieurs usages (RGB et YUV par exemple). Un magnétoscope par exemple ne sortira généralement jamais un signal RGB, mais simplement composite. Un téléviseur lui, en revanche, acceptera peut-être tous types de signaux arrivant par la péritel et permettra donc à un équipement fournissant un signal RGB Sync-on-composite d'afficher une image de qualité. Et ce, même si le téléviseur en question ne dispose pas d'une connectique spécifique RVA ou BNC. Bien entendu, ceci est totalement dépendant du fabricant et du modèle.

Cette non-disponibilité de tous les signaux est également valable pour les upscalers, qu'il s'agisse de modèles bas de gamme à quelques 30€ ne supportant généralement que le composite (en double, une fois en cinch et une fois en péritel), ou des choses plus spécialisées comme l'OSSC (pas de Y/C) ou le Framemeister (uniquement RGB + CSYNC) sur le connecteur RGB via un adaptateur SCART.

À ce propos justement, le Framemeister est livré par défaut avec ce qui semble être un adaptateur péritel vers miniDIN-8. Ce n'est pas le cas ! Il s'agit en réalité d'un autre type de connexion **mécaniquement** compatible avec une prise péritel, mais utilisant un brochage différent : JP-21 (appelé RGB-21 au Japon et en Corée). Bien que n'ayant pas été aussi populaires en Asie que la péritel en Europe, un certain nombre de mods et de hacks pour les consoles Sega et Nintendo font référence à cette norme. Les signaux sont les mêmes, rouge, vert, bleu, synchronisation, audio gauche et audio droite, mais les broches sur péritel sont respectivement 12, 11, 7, 20, 6, et 2, alors que sur JP-21 elles sont 15, 19, 20, 9, 1 et 5.

POUR FINIR

Le but de cet article n'était pas de vous présenter une ou plusieurs façons de connecter votre ordinateur ou votre console vintage à votre téléviseur ou votre upscaler, mais de vous donner les clés vous permettant de comprendre quelles sorties utilisent quels signaux et l'importance de chaque norme en termes de résultats.

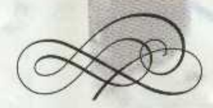
Vous l'avez sans doute compris, les différentes possibilités ont été listées ici par ordre croissant de qualité à attendre ou espérer. Mais ceci n'est pas le seul critère déterminant. La qualité des câbles est importante, tout autant que le type de matériel utilisé pour convertir le signal, qu'il s'agisse d'un équipement dédié ou de celui intégré dans votre TV.

Imaginez cela comme de la cuisine. Beaucoup vous diront à propos d'un vin peu satisfaisant que ce n'est pas bien grave et qu'il pourra être utilisé pour faire une sauce. Je n'adhère pas à ce concept, car un vin passable, en toute logique, vous donnera une sauce passable. L'image obtenue sur un écran est un peu comme une sauce, il faut que l'ingrédient de base soit de qualité. Vous pouvez tout gâcher en le cuisinant n'importe comment, certes, mais inversement vous n'obtiendrez jamais quelque chose d'excellent avec une base intrinsèquement médiocre, même en étant le plus grand chef au monde.

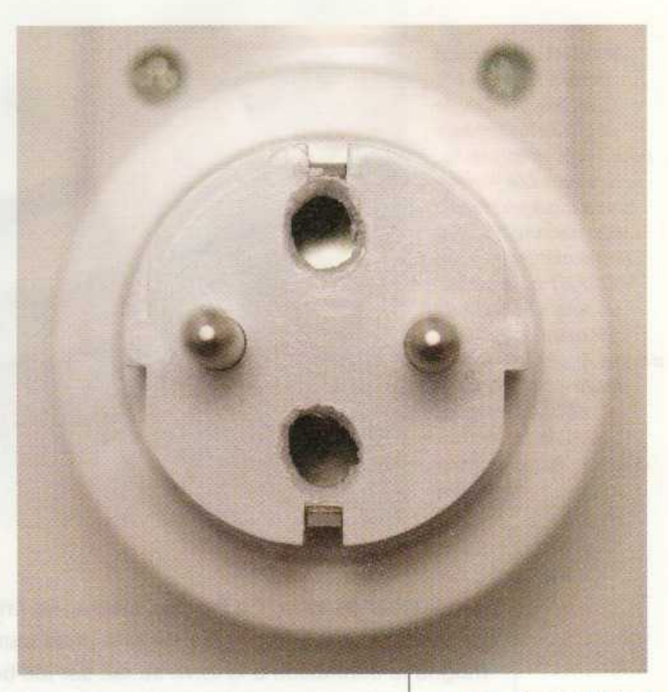
J'espère donc que cette énumération détaillée vous permettra de choisir votre ingrédient principal en fonction du résultat visé, de sa disponibilité et de sa difficulté à être obtenu. Et ce, quelle que soit votre plateforme fétiche. DB

LA TÉLÉCOMMANDE ARDUINO, LE RETOUR VERSION ESP8266 !

Denis Bodor



Dans le douzième numéro du magazine, j'avais décrit la construction d'une télécommande radio Arduino pour prises de courant, reposant sur une puce Silicon Labs si4021, ou plus exactement un gros hack du circuit provenant d'une autre télécommande. Il est temps de revisiter le sujet, en mieux, en beaucoup, beaucoup mieux et à la sauce ESP8266, car non seulement les outils ont sensiblement évolué, mais surtout, une nouvelle « génération » de prises télécommandées se popularise et s'avère fort intéressante d'un point de vue technique et pratique.



Le principal défaut de ces prises télécommandées programmables est leur utilisation de la norme dite « type F » correspondant à un standard électrique allemand et non français (type E). Elles ne peuvent donc être utilisées avec une prise murale française sans modification et celle-ci n'assure pas la mise à la terre. Il est alors, du moins en ce qui me concerne, hors de question de les utiliser pour autre chose que des luminaires ne disposant de toute façon pas de prise terre.

Commençons par le plus important, ces nouvelles prises (du moins nouvelles pour moi) ne sont pas d'une marque précise très connue, mais sont vendues sous de nombreuses désignations. J'ai acheté les miennes sur eBay auprès d'un vendeur appelé *xdeal2013* et via une annonce ayant pour titre « *3 Pack Wireless Remote Control Power Outlet Light Switch Plug Socket* ». L'ensemble était proposé sous la forme d'un lot de trois prises et d'une télécommande, pour 17,98€ port offert, et les produits sont arrivés depuis Shenzhen en quelques 9 jours. « Les », parce que j'ai commandé trois lots, soit 9 prises et 3 télécommandes...

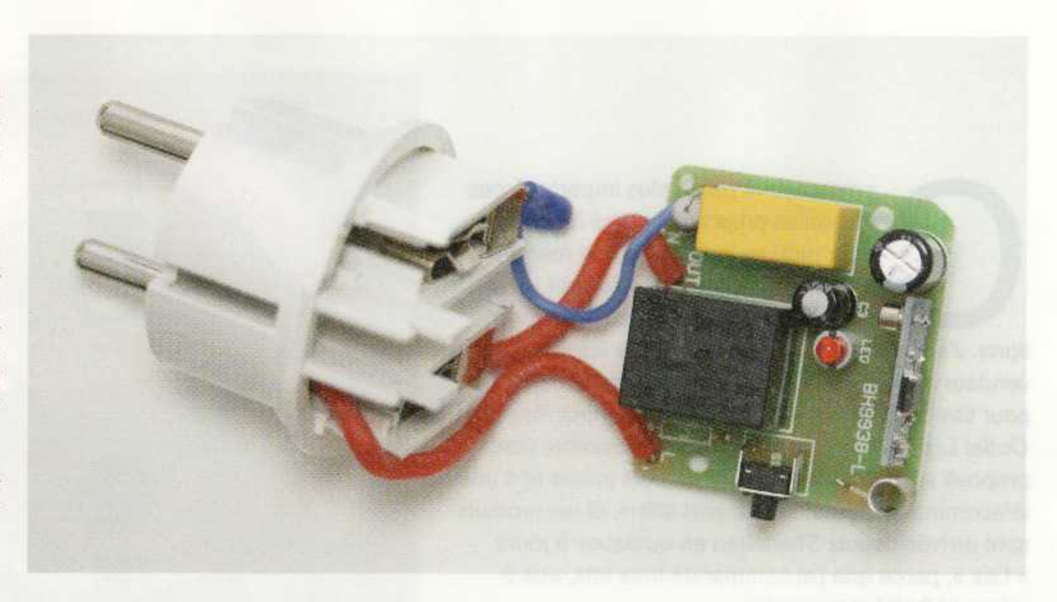
1. LES PRISES TÉLÉCOMMANDÉES

Mais pourquoi tant de prises ? La réponse est toute simple : celles-ci sont **programmables**. En d'autres termes, chaque prise peut être configurée pour répondre à une paire de boutons (on/off) de n'importe laquelle des télécommandes associée. Mieux encore, cette programmation n'est pas exclusive et il est possible de faire en sorte qu'une prise accepte les ordres de n'importe quelle paire de boutons de n'importe quelle télécommande. Résumé plus simplement, vous pouvez commander plusieurs prises avec une paire de boutons, mais aussi une prise avec de multiples paires de boutons et donc plusieurs télécommandes.

Cette fonctionnalité de programmation est accessible à l'aide un simple bouton placé sur le côté de la prise et cette caractéristique fait partie des éléments qui vous permettront de reconnaître le produit dans vos recherches. En effet, en cherchant « Wireless Remote Outlet » sur eBay, DX, Banggood ou autre, vous risquez de voir apparaître une collection impressionnante de modèles. J'ai eu initialement connaissance de ces produits via une vidéo YouTube (de bitluni's lab) présentant un montage similaire à celui de *Hackable n°12*, mais précisant qu'il était possible de commander quelques 1048576 prises. D'un naturel assez méfiant et ce chiffre étant relativement étonnant, j'ai donc recherché le même type de matériel en me basant principalement sur l'aspect du produit.

Le matériel réceptionné cependant possède un nom moulé dans le plastique, « Intertek », mais l'ajout de ce terme dans une recherche n'apporte aucune amélioration quant aux résultats affichés. Si le vendeur que je mentionne ne vend plus le produit au moment où vous lirez cet article, vous devrez donc procéder comme moi en misant sur l'aspect visuel et, de ce fait, en acceptant une certaine part de risque dans la démarche.

La qualité générale du produit est assez correcte à l'exception de la télécommande dont la trappe pour la pile type A23 12V est une véritable horreur. À noter également que le vendeur proposait deux déclinaisons pour le produit : « US plug » et « EU plug », prise US ou prise européenne. Or la version EU est en réalité adaptée aux prises allemandes (type F, alias « Schuko ») et non françaises (type E). Mécaniquement en partie compatibles, les deux types se différencient par leur



Le circuit se trouvant dans les prises est relativement simple et utilise un module pour la réception radio. L'ensemble repose sur la mise en œuvre d'un microcontrôleur (sans marquage), un MOSFET pour piloter un relai et quelques éléments passifs. Il devrait être relativement facile de réutiliser le circuit complet ou le module radio pour diverses réalisations si nécessaire.

connecteur de terre. En France, celui-ci se présente sous la forme d'un connecteur mâle sur une prise murale, en Allemagne la connexion à la terre se fait sur les bords, perpendiculairement aux connecteurs de la phase et du neutre.

Il est donc possible de brancher un cordon d'alimentation français sur une prise type F, mais pas l'inverse. La plupart des cordons d'alimentation, comme ceux des PC, utilisent la norme CEE 7/7, avec un trou pour la terre française et des connecteurs en bordure pour la terre allemande. En résumé, vous ne pouvez pas brancher ces nouvelles prises télécommandées au mur sans faire un trou dans le plastique. Dans le cas présent, c'est parfaitement possible du fait de la construction du produit et de la présence de séparateurs en plastique entre le centre et les côtés de la structure. Mais **ATTENTION** ! Ceci n'offrira aucune mise à la terre de l'équipement branché sur la prise télécommandée. Une autre solution, plus sûre, serait d'utiliser un adaptateur dédié entre les deux types (que je n'ai pas réussi à trouver pour l'instant).

Personnellement je ne compte brancher, sur ces prises, que des luminaires ne disposant de toute façon pas de connexion à la terre (type C, alias *Europlug*) ou utilisant des blocs d'alimentation également dépourvus de cette connexion. Ceci ne pose donc pas de problème, mais c'est quelque chose qu'il est important de signaler et de garder à l'esprit.

2. UTILISATION STANDARD DES PRISES TÉLÉCOMMANDÉES

Pour programmer les prises, rien de de plus simple, il suffit de suivre les indications données dans la documentation, enfin... la simple feuille A5 rédigée très classiquement en *chinglish*. Branchez la prise au mur, appuyez sur le bouton jusqu'à ce que la led de la prise clignote rapidement, relâchez le bouton et pressez un bouton « ON » ou « OFF » d'une télécommande. La led clignote brièvement et la prise accepte maintenant cette paire de boutons en guise de contrôle.

Tout naturellement, afin de s'accoutumer à la procédure, on s'amuse ainsi un temps avec un lot de trois prises et une télécommande avec différentes combinaisons : une paire de boutons pour plusieurs prises, tous les boutons d'une télécommande qui contrôle une prise, etc. Et là, arrive forcément le moment où on se dit « *ok, assez joué, revenons à l'état initial* » et là... on se retrouve comme un imbécile, puisque la documentation ne parle pas du tout de l'effacement ou de la réinitialisation !

Après une bonne dizaine de minutes à essayer les combinaisons possibles, rester appuyé sur le bouton plus longtemps, activer le mode d'apprentissage, mais sans utiliser la télécommande, appuyer sur le bouton de la télécommande

puis activer le mode apprentissage... on arrive finalement à trouver la technique oubliée par la documentation. Et je vous la livre en exclusivité dans cet article : il faut enfoncer le bouton de la prise sans l'avoir branchée, faire un tour sur soi-même sur un pied avec la prise en main (optionnel), et brancher la prise au mur tout en maintenant le bouton enfoncé. La led clignote quatre fois, indiquant que la mémoire est totalement effacée. Dès lors, la prise ne répondra plus à aucune télécommande, même celle initialement livrée avec le lot et on pourra la reprogrammer comme détaillé précédemment.

Il apparaît donc clairement que le microcontrôleur embarqué dans la prise enregistre dans sa mémoire flash ou EEPROM (la programmation perdure même si la prise n'est pas alimentée) la liste des codes de télécommandes auxquels il doit obéir. J'ignore quel est le nombre maximum de codes mémorisables par une prise, mais mes essais ont montré qu'au minimum 4 codes (ou paires de boutons) peuvent être mémorisés. Un tel code se résumant, comme nous allons le voir, à trois malheureux octets (24 bits), il est tout à fait possible que ce maximum s'exprime en dizaines de codes. À titre d'exemple, même l'ultra économique microcontrôleur Atmel ATtiny25 dispose de 128 octets d'EEPROM, de quoi stocker plus d'une quarantaine de codes de 3 octets.

Cette reprogrammation des prises est furieusement intéressante, car on peut ainsi envisager une architecture personnalisée. Je vais ici égoïstement décrire mon propre agencement, et donc mon projet, mais vous pourrez sans peine assez facilement l'adapter à votre propre installation...

3. L'OBJECTIF FINAL : CONFORT ABSOLU

La « pièce à vivre » (et surtout à travailler) de mon petit nid douillet est un double séjour (ou salon/salle à manger) comme l'aiment à le désigner les agences immobilières. Il se compose donc de deux parties qui, fut un temps, était des pièces individuelles (bâtisse de 1806). Il serait abusif de désigner cela comme « un salon et une salle à manger » car, pour les yeux profanes, tout ceci ressemble à un amalgame entre des bureaux, un atelier d'électronique et un musée (mais il y a des fauteuils, donc c'est bien un salon).

Quoi qu'il en soit, l'aspect luminaire, lui, est plus explicite. Il y a d'une part un éclairage à tubes fluo composé de 2 fois 4 tubes pour travailler, bricoler, bidouiller, démonter, rénover, etc. Et d'autre part, un ensemble plus tamisé composé d'un éclairage au sol, d'une barre à leds au-dessus d'une bibliothèque, d'une lampe champignons en pâte de verre et de l'éclairage interne d'un TARDIS fait maison à l'échelle 1/1 (un demi-TARDIS qui sert de porte en réalité, mais il est donc effectivement plus grand à l'intérieur qu'à l'extérieur).

Nous avons donc en main un ensemble de luminaires à contrôler selon la configuration suivante :

- mode « travail » : l'ensemble des 8 tubes fluo pour les deux parties de la pièce ;
- mode « confort » : sol, barre de leds, champignon, TARDIS ;
- mode « demi-travail » : 4 tubes fluo côté bureau principal.

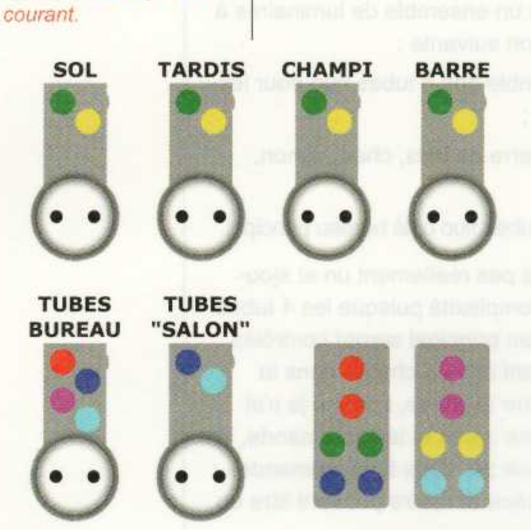
Ce dernier mode n'en sera pas réellement un et ajoutera simplement un peu de complexité puisque les 4 tubes qui ne sont pas du côté bureau principal seront contrôlés indépendamment, tout en étant pris en charge dans le mode travail. Et pour compléter la chose, comme je n'ai pas envie de courir sans cesse après la télécommande, le système doit pouvoir être piloté par deux télécommandes : une restant à l'entrée de la pièce et l'autre pouvant être un peu plus nomade.

Ce n'est pas tout. L'idée est de pousser le vice encore davantage car, vous en conviendrez avec moi, devoir se saisir d'une télécommande alors qu'on se trouve devant son PC n'a pas de sens (il faudrait retirer sa main du clavier ou de la souris). il convient donc de faire en sorte que le PC puisse également faire office de télécommande pour le système. Je pense en particulier à la situation selon laquelle je bascule du monde travail au mode confort, après avoir fini de rédiger un code ou un article, par exemple, et où je m'accorde un peu de détente en regardant quelques vidéos de mes chaînes YouTube préférées.

Un bouton dans le navigateur (Chromium) devra donc permettre de changer de mode d'éclairage sans toucher à une télécommande physique. Mieux encore, ceci devra se faire via une interface web, et sera donc également implicitement accessible par d'autres biais (smartphone par exemple) et indépendamment du navigateur, du système d'exploitation ou du type d'ordinateur.

La configuration des prises dans ce schéma ne présente pas de grande difficulté. Il s'agit principalement de ne simplement pas se mélanger les pinceaux et, pour cela, l'usage d'une technologie adaptée, fiable et éprouvée s'avère indispensable : un papier et un crayon. Chacune des six prises est préalablement « mise à zéro » selon la méthode décrite plus haut puis on programme séquentiellement quelle prise s'utilise avec quelle télécommande. J'ai résumé la logique complète sous la forme d'un schéma présenté ici via des codes de couleurs, mais le brouillon papier était similaire tout en utilisant des lettres.

Une fois l'ensemble reprogrammé, les deux télécommandes agissent de la même manière sur les groupes de prises. Selon le schéma, les boutons rouges ou magenta contrôlent l'éclairage fluo de la moitié de la pièce, les boutons bleus ou cyan font de même pour l'ensemble des tubes fluo et les boutons verts ou jaunes pilotent l'ensemble des luminaires de l'ambiance tamisée.



L'architecture générale de l'éclairage repose sur une reprogrammation des prises de façon à ce que chacune d'elles joue un rôle bien particulier dans l'ensemble. On obtient ainsi deux ambiances différentes, une avec un éclairage fort pour travailler efficacement et une lumière plus tamisée pour la détente, tout en permettant le contrôle individuel de l'une des prises, pour affiner et économiser un peu de courant.

Jusqu'ici nous n'avons fait qu'utiliser les fonctionnalités tout à fait standards de ces équipements. Notez cependant qu'avec des prises télécommandées classiques, avec une sélection de canal sur les prises et les télécommandes, il ne serait pas possible de reproduire cette architecture. Du moins pas sans avoir à jouer avec le sélecteur de canal des télécommandes qui, la plupart du temps, prend la forme d'interrupteurs miniatures.

Pour étendre le fonctionnement et le rendre automatisable, nous devons maintenant nous pencher sur la simulation d'une télécommande via un montage électronique et pour cela, comprendre le fonctionnement du produit.

4. PRENDRE LE CONTRÔLE AVEC L'ESP8266

La première approche que nous allons utiliser consiste à mettre en œuvre un circuit intégré Silicon Labs si4021 tel que nous l'avions vu ensemble dans le numéro 12 puis, plus tard, dans le numéro 19. Le si4021 est un transmetteur universel pour les bandes de fréquences dites ISM (industriel, scientifique et médical) capable de travailler sur 433, 868,

La minuscule documentation du HS1527 résume toutes les informations qui nous sont nécessaires et présente même un circuit typique d'utilisation. Ce composant n'a pas besoin de microcontrôleur et on distingue une séparation en trois parties avec au centre toute la logique intégrée dans le HS1527, à droite l'aspect interface et à gauche le circuit radio/HF.

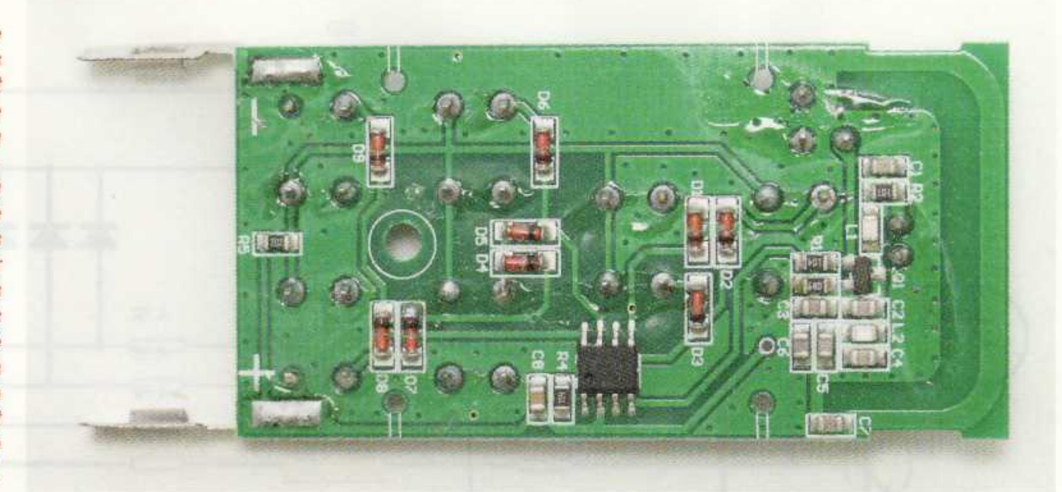
et 915 MHz (cette dernière bande de fréquences ne fait pas partie des bandes ISM en Europe).

Nos prises utilisent la fréquence de 433,92 Mhz, parfaitement à portée du si4021. Celui-ci est qualifié d'universel dans la documentation Silicon Labs, non seulement en raison de l'utilisation possible de plusieurs bandes de fréquences, mais aussi et surtout par le fait qu'il se pilote via un bus SPI (données du maître vers l'esclave, signal d'horloge et broche d'asservissement). Ce moyen de communication permet de configurer le composant en inscrivant des valeurs dans ses registres afin de régler la fréquence utilisée, la puissance de l'émission, la modulation (OOK ou FSK) ou encore la gestion d'énergie.

Il s'agit donc d'un composant polyvalent allant bien au-delà de la simple construction d'une télécommande et il ne s'agit que d'un périphérique. Le désavantage de cette solution est qu'il faut donc impérativement un microcontrôleur pour le configurer et le piloter. Le point positif lui, est qu'il est possible de faire absolument tout ce qu'on veut sans être « coincé » dans une unique forme d'utilisation.

Le composant se trouvant dans les télécommandes fraîchement acquises n'est pas un si4021. Un rapide démontage de l'une d'entre elles révèle qu'il n'y a qu'un circuit intégré présent sur le circuit et non deux. Ce circuit est un HS1527 de *Silvan Chip Electronics Tech*, marqué eV1527, et peut être vu comme l'exact opposé du si4021. C'est une solution « tout intégrée » d'encodage n'incluant aucune partie radio/fréquence. Le HS1527 est conçu pour lire l'entrée de 4 de ses broches (K0 à K3) et il produit une série de 24 bits précédés d'un préambule sur sa sortie TXD).

Ces changements d'états sur TXD sont alors utilisés par un circuit externe pour moduler un signal qui est alors transmis par les airs. Le HS1527 peut générer une oscillation qui, via un transistor NPN, peut être combinée à la série de bits pour produire une modulation OOK (*On Off Keying* ou en français « tout-ou-rien »). Dans le cas de nos télécommandes, la broche du HS1527 permettant de configurer la fréquence d'oscillation n'est pas utilisée et une autre partie du circuit, construite autour d'un oscillateur à quartz et d'un transistor, est utilisée pour générer le signal. Ceci peut paraître anodin, mais cette séparation entre l'encodage et la modulation est une véritable aubaine en ce qui nous concerne.



La télécommande livrée avec les prises utilise un seul circuit intégré, HS1527, se chargeant de toute la partie « intelligente ». Une utilisation astucieuse d'une dizaine de diodes permet de combiner l'état des 6 boutons pour former 3 bits utilisés en entrée de l'encodeur. Toute la partie à droite de la photo est le circuit radio chargé de l'émission du signal.

Quoi qu'il en soit, la volumineuse (sic) documentation du HS1527, qui se résume à deux pages, nous apporte une autre information capitale : l'organisation des 24 bits qui composent un message. Nous avons 20 bits fixes provenant directement de la mémoire de la puce, suivis de 4 bits représentant l'état combiné des entrées K0 à K3 (2 puissance 4 = 16 combinaisons possibles). Ces 4 derniers bits nous informent donc de l'état des boutons de la télécommande, mais les 20 précédents sont probablement uniques à chaque télécommande (2 puissance 20 = 1048576 possibilités). Je dis « probablement », car la documentation décrit le HS1527 comme un « OTP Encoder », OTP signifiant « *One Time Programmable* » et donc avec une programmation certainement faite en usine.

Chaque télécommande envoie donc des messages de 24 bits, avec 20 bits propres à chacune d'elle et 4 bits décrivant l'état des 6 boutons. Tout ce que nous avons à faire c'est obtenir quelques messages pour savoir comment sont utilisés ces bits.

4.1 Avec un récepteur RTL-SDR

Comme nous l'avons fait de nombreuses fois dans le magazine, l'approche pratique la plus évidente pour obtenir l'information dont nous avons besoin repose sur l'utilisation d'un récepteur radio logiciel. Pour rappel, une gamme complète de périphériques USB très abordables (entre 20€ et 30€) est aujourd'hui disponible et permet de recevoir toutes sortes d'émissions radio. Initialement issu d'un hack reposant sur l'utilisation d'un récepteur DVB-T (TNT), tout un écosystème matériel et logiciel a vu le jour et s'est bonifié au fil des années.

Un récepteur SDR de ce type, également connu sous la désignation « RTL-SDR » est une simple association d'un tuner permettant la réception de signaux à une fréquence donnée et d'un convertisseur analogique-numérique transformant les signaux dans un format qu'un ordinateur peut traiter. Le nom RTL-SDR découle directement du nom d'un des deux composants généralement utilisés : le RTL2832 de Realtek.

Une vaste gamme d'applications et d'outils est disponible aussi bien pour Windows que pour macOS, mais aussi, et surtout pour GNU/Linux. C'est généralement avec ce système que les petits outils les plus intéressants sont universellement compatibles en premier lieu (même si souvent des déclinaisons Windows existent). L'outil qui nous occupera ici est `rtl_433`. Il s'agit d'un récepteur modulaire capable de décoder les données provenant de différents équipements fonctionnant généralement à 433,92 Mhz : télécommandes, sondes de température, capteurs de pression des pneus (TPMS), stations météo, etc.

Les prises concernées par cet article ne sont pour l'instant pas prises en charge par `rtl_433` (peut-être dans le futur si j'ai un peu de temps), mais ceci n'a aucune importance. L'une des fonctionnalités intégrées à `rtl_433` concerne l'analyse de nouveaux signaux avec un traitement quasi automatique des impulsions. L'objet de `rtl_433` est de décoder les informations, mais la présentation d'une série de bits d'un signal reçu nous sera bien suffisante.

Pour installer rtl_433 sur un PC GNU/Linux ou une Raspberry Pi, il vous faudra tout d'abord installer les paquets indispensables à sa construction/compilation : libtool, libusb-1.0.0-dev, librtlsdr-dev, rtl-sdr, build-essential, autoconf, cmake, pkg-config et git. Ceci fait, nous pourrons obtenir les sources de rtl_433 depuis GitHub avec git clone https://github.com/merbanan/rtl_433.git.

La construction à proprement parler se fera avec cette série de commandes :

```
$ cd rtl_433
$ mkdir build
$ cmake ../
$ make
$ cd src
```

Vous trouverez alors, dans le répertoire courant, le programme rtl_433 que vous pourrez lancer avec :

```
$ ./rtl_433 -a
[...]
Found 1 device(s)

trying device  0:  Realtek, NESDR_SMArt, SN: 00000001
Found Rafael Micro R820T tuner
Using device 0: Generic RTL2832U OEM
Exact sample rate is: 250000.000414 Hz
Sample rate set to 250000.
Bit detection level set to 0 (Auto).
Tuner gain set to Auto.
Reading samples in async mode...
Tuned to 433920000 Hz.
[...]
```

L'option -a est celle permettant d'activer le mode « analyse » et l'outil est alors à l'écoute de tous signaux pouvant être réceptionnés sur la fréquence par défaut de 433,92 Mhz. Il nous suffit alors d'utiliser un des boutons de la télécommande pour voir apparaître quelque chose comme :

```
*** signal_start = 335664, signal_end = 420208
signal_len = 84544,  pulses = 311
Iteration 1. t: 67    min: 27 (200)    max: 108 (111)    delta 29
Iteration 2. t: 67    min: 27 (200)    max: 108 (111)    delta 0
Pulse coding: Short pulse length 27 - Long pulse length 108

Short distance: 53, long distance: 133, packet distance: 1257

p_limit: 67
bitbuffer:: Number of rows: 14
[00] {1} 00 : 0
[01] {25} 28 a0 dc 00 : 00101000 10100000 11011100 0
[02] {25} 28 a0 dc 00 : 00101000 10100000 11011100 0
[03] {25} 28 a0 dc 00 : 00101000 10100000 11011100 0
[...]
```

En plus des informations de durée des impulsions, nous avons sous nos yeux les 24 bits dont nous avons besoin (le dernier `0` peut être ignoré). En répétant l'opération avec les six boutons d'une télécommande, nous pouvons en déduire l'information attendue :

```
B1 ON  : 00101000 10100000 11011100
B1 OFF : 00101000 10100000 11010100
B2 ON  : 00101000 10100000 11011010
B2 OFF : 00101000 10100000 11010010
B3 ON  : 00101000 10100000 11011001
B3 OFF : 00101000 10100000 11010001
```

Les quatre derniers bits de chaque message sont, comme supposé, variables. `00101000101000001101` correspond donc aux 20 bits uniques de la puce HS1527 de la télécommande. On remarque également dans les six messages un motif récurrent : le premier des quatre bits variables est à `1` pour ON et à `0` pour OFF. Le reste des bits correspond à chaque paire de boutons : `100` pour la première, `010` pour la seconde et `001` pour la troisième. Ce qui est confirmé par une étude du circuit de la télécommande reposant sur l'utilisation relativement ingénieuse de boutons poussoirs couplés à des diodes (qui semblent être des 1N4148).

Nous avons donc trouvé ce que nous cherchions, le 21ème bit change l'état de la prise. Mais la vidéo de bitluni's lab se trompe, ce n'est pas 1048576 prises qui peuvent être commandées ainsi, mais 3145728 ! 1048576 identifiants uniques certes, mais fois trois canaux.

Pas de récepteur RTL-SDR ?

L'utilisation d'un récepteur RTL-SDR n'est pas une absolue nécessité si vous comptez réaliser une installation similaire. Nous avons pris connaissance à présent de l'organisation des bits du message et en particulier de l'usage réservé au 21ème bit. Pour simuler une télécommande, il ne sera donc pas absolument nécessaire de capturer un signal d'une télécommande existante. Nous pouvons tout simplement forger arbitrairement un message selon la même structure et programmer une ou plusieurs prises pour y réagir.

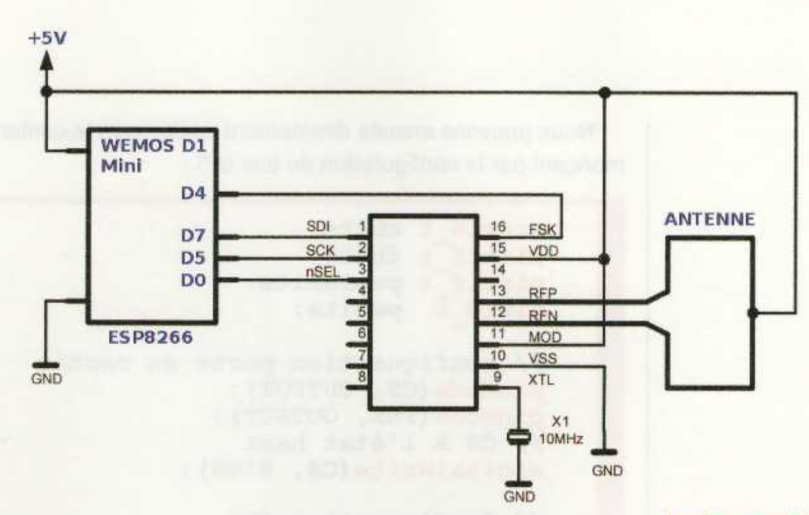
Mais si vous tenez absolument à copier un message existant sans disposer d'un récepteur SDR, il est également possible de lire l'état de la broche TXD du HS1527 lors de l'utilisation d'un des boutons. Ceci suppose le démontage de la télécommande et l'utilisation d'un analyseur logique (type clone de Saleae Logic + Sigrok) ou d'un oscilloscope, mais c'est tout à fait faisable.

4.2 Utilisation du si4021

Le circuit intégré si4021 se trouve facilement chez les détaillants de composants et sur des sites comme eBay. La difficulté de mise en œuvre cependant n'est pas l'approvisionnement du composant, mais son format. Celui-ci n'est, en effet, uniquement disponible sous forme de paquet TSSOP16 (pour *Thin-Shrink Small Outline Package* 16 broches), ce qui impose l'utilisation d'un circuit imprimé permettant l'adaptation au format DIP16 et donc une utilisation sur platine à essais ou avec des câbles *jumpers* au pas de 2,54 mm.

Ce n'est pas tout, le si4021 nécessite l'ajout d'un quartz à 10 Mhz afin de pouvoir générer la ou les fréquences qui seront utilisées pour l'émission, ainsi qu'une antenne. Cette dernière pourra être très simplement construite avec un câble monobrin, mais sa forme et sa taille doivent être adaptées à la fréquence utilisée. La construction d'antennes est une science obscure, sinon de la pure magie noire en ce qui me concerne, et même si bricoler quelque chose de fonctionnel est tout à fait possible, il est fort peu probable que vous arriviez à quelque chose d'optimal sans études approfondies du domaine.

Pour ma part, j'ai tout simplement pris exemple sur une antenne PCB d'une télécommande utilisant



La documentation officielle du si4021 propose une mise en œuvre typique. Ceci en est une version simplifiée et adaptée à la connexion avec une carte Wemos D1. Vous pouvez voir ici la structure particulière de l'antenne qu'il sera nécessaire de reproduire avec les moyens du bord.

un si4021 à 433,92 Mhz. Avec du câble monobrin, j'ai ainsi formé un rectangle de 30 mm sur 20 mm, divisé verticalement en son milieu pour la connexion à la tension d'alimentation, les deux extrémités étant reliées aux broches RFN et RFP du si4021.

La connexion avec le module ESP8266, ici un clone Wemos D1 Mini, se résume à :

- 1 MOSI : données du maître vers l'esclave sur D7 ;
- 2 SCK : signal d'horloge sur D5 ;
- 3 nSEL : asservissement (/CS) sur D0 ;
- 9 XTL : quartz en série avec la masse ;
- 10 VSS : masse ;
- 11 MOD : au +5V (le si4021 est mode microcontrôleur) ;
- 12 RFN : antenne pôle positif ;
- 13 RFP : antenne pôle négatif ;
- 15 VDD : tension d'alimentation +5V ;
- 16 FSK : signal en mode OOK, sur D4.

Le contrôle du si4021 par l'ESP8266 se fait via la connexion SPI pour la configuration et via une unique broche (FSK) pour l'activation. Le croquis final du projet, dans cette déclinaison, se compose du croquis principal, mais aussi des fichiers `si4021.cpp` et `si4021.h` regroupant les macros et fonctions propres à la configuration du composant. Je ne rentrerai pas dans le détail du contenu de ces fichiers ici faute de place et me contenterai d'expliquer l'utilisation de ces fonctions spécifiquement développées. J'invite le lecteur curieux à récupérer l'ensemble dans le dépôt GitHub associé à ce numéro (**https://github.com/Hackable-magazine/Hackable24**) et à consulter les commentaires dans ces fichiers.

Pour configurer notre si4021, nous utiliserons donc directement le bus SPI de l'ESP8266, mais la partie modulation sera le travail de la bibliothèque *rc-switch* développée par Suat Özgür (alias sui77) et disponible sur **https://github.com/sui77/rc-switch**. C'est cette bibliothèque qui se chargera de transformer notre série de `0` et de `1` en un signal qui sera modulé (OOK) puis émis par le si4021. Mais avant toutes choses, commençons par configurer la base avec l'inclusion des fichiers d'entête et la définition des macros et variables :

```
#include <SPI.h>

#include "si4021.h"

#define CS D0
#define FSK D4

struct cfg {
  float freq;
  unsigned int attenuation;
} maconfig = { 433.92, PS_0DB };
```

Les paramètres devant être utilisés par le si4021 prennent ici la forme d'une structure regroupant la fréquence à utiliser (`433.92`) et l'éventuelle atténuation à mettre en œuvre au niveau de l'amplificateur intégré au si4021 (`PS_0DB` est défini dans `si4021.h` et correspondant à aucune atténuation).

Nous pouvons ensuite directement sauter vers le contenu de la fonction **setup()**, en commençant par la configuration du bus SPI :

```
uint16_t cbits;
uint16_t fbits;
uint16_t pwmanbits;
uint8_t  pwbits;

// configuration ports en sortie
pinMode(CS, OUTPUT);
pinMode(FSK, OUTPUT);
// CS à l'état haut
digitalWrite(CS, HIGH);

// Configuration SPI
SPI.begin();
SPI.setClockDivider(SPI_CLOCK_DIV16);
SPI.setDataMode(SPI_MODE0);
SPI.setBitOrder(MSBFIRST);
```

Ceci fait, nous pouvons ensuite nous pencher sur la configuration du si4021 lui-même, via l'utilisation des fonctions définies dans **si4021.cpp** :

```
// configuration de la fréquence sur 433.92 MHz
Serial.println("si4021: Configuration Setting");
if(!(cbits = confset(maconfig.freq))) {
  Serial.println("ERROR : bad config!");
  while(1);
} else {
  sendToSi4021(CS, cbits);
}
Serial.println("si4021: Frequency Setting");
if(!(fbits = freqset(maconfig.freq))) {
  Serial.println("ERROR : bad frequency!");
} else {
  sendToSi4021(CS, fbits);
}

// Configuration de l'émission
Serial.println("si4021: Power Setting");
if(!(pwbits = powerset(maconfig.attenuation))) {
  Serial.println("ERROR : bad power settings!");
} else {
  sendToSi4021(CS, pwbits);
}

// Configuration de l'alimentation
Serial.println("si4021: Power Management");
if(!(pwmanbits = powerman())) {
  Serial.println("ERROR : bad power management settings!");
} else {
  sendToSi4021(CS, pwmanbits);
}
```

Le si4021 se configure en précisant des valeurs dans différents registres qu'il met à disposition. Les fonctions comme **confset()** par exemple, utilisent la structure, où nous avons stocké nos paramètres, pour calculer l'état de chaque bit dans ces registres. La valeur pour chaque registre est ensuite inscrite en utilisant la fonction **sendToSi4021()** prenant en argument la broche associée à la ligne /CS ainsi que la valeur du registre (qui intègre également la désignation du registre concerné).

Arrivé là dans la fonction **setup()**, le si4021 est prêt à être utilisé et est d'ores et déjà en marche. Il est configuré pour émettre en fonction de l'état de sa broche FSK et c'est précisément ce contrôle que nous passons à la bibliothèque *rc-switch*. Après avoir inclus le fichier d'en-tête adéquat, nous commençons par déclarer un objet dont les méthodes nous permettront la commande de l'émission :

```
#include <RCSwitch.h>

RCSwitch mySwitch = RCSwitch();
```

Mais avant de pouvoir faire cela, nous devons également configurer la modulation et divers paramètres :

```
/***** Configuration RCSwitch OOK *****/
// On utilise la sortie FSK (broche 8)
mySwitch.enableTransmit(FSK);
// Option : durée de l'impulsion
// mySwitch.setPulseLength(125);
// Option : protocole (1 fonctionne la plupart du temps)
mySwitch.setProtocol(1);
// Option : nombre de répétitions du message
mySwitch.setRepeatTransmit(5);
```

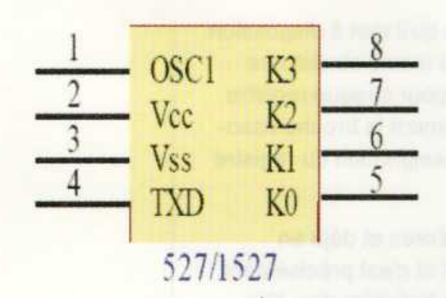
Tout est maintenant prêt pour les premiers tests et nous pouvons vérifier très simplement que l'ensemble fonctionne en utilisant par exemple **mySwitch.send("001010001010000011011100")**. La méthode **send()** prend en argument une chaîne de caractères représentant le message à envoyer et donc les 24 bits correspondants à l'émission provoquée par l'appui d'un des boutons de la télécommande simulée. Normalement, ceci devrait permettre d'activer l'une des prises programmées pour réagir à ce message.

À ce stade, nous pourrions nous asseoir sur nos lauriers et finaliser rapidement le croquis comme nous l'avions fait dans un précédent numéro. Mais nous voulons aller plus loin en offrant une véritable interaction ergonomique. Nous devons donc ajouter les éléments de connectivité Wifi et de quoi permettre le contrôle via un navigateur web.

Mais avant cela, faisons un petit détour dans le domaine du hacking...

4.3 Le si4021 est-il vraiment nécessaire ?

J'avoue que mon approche consistant à utiliser un si4021 est la première qui m'est venue à l'esprit et que j'ai implémentée. Ce n'est qu'ensuite, en étudiant de plus près l'une des télécommandes qu'une solution bien plus amusante et plus économique s'est fait jour : nous pouvons parfaitement nous passer de si4021 et de bricolage d'antenne, et hacker directement l'une des télécommandes.



La documentation du HS1527 précise son brochage. Pour remplacer le composant dans la télécommande par notre montage à base d'ESP8266, tout ce que nous avons à faire est de relier les masses ensemble et connecter l'emplacement pour la broche TXD à la broche D4 de la carte Wemos D1 mini. Notez que le HS1527 possède une marque en relief pour indiquer la broche 1 et que l'emplacement sur le circuit est sérigraphié avec un profil montrant une encoche pour préciser l'orientation du composant.

La première chose à laquelle on pense sur le sujet est de, tout simplement, simuler l'utilisation des boutons. En effet, cela peut fonctionner, mais, avouons-le, c'est une approche qui n'est pas très élégante, voire totalement bricolo-immonde. Allons, allons, nous sommes bien plus joueurs que cela voyons !

Si j'ai parlé du HS1527, ce n'est pas par hasard. En regardant de plus près le circuit proposé par sa documentation et l'implémentation qui en est faite dans la télécommande, on se rend compte que la broche OSC1 n'est pas utilisée et que ce n'est pas le HS1527 qui génère la fréquence de 433,92 Mhz. Ceci est la tâche d'une autre partie du circuit, le HS1527 ne faisant que fournir, en version encodée, les 20 bits d'identification et les 4 bits variables du message.

En d'autres termes, si nous retirons le HS1527 d'une télécommande, nous pouvons moduler le signal nous-mêmes avec une totale liberté. Ceci nous débarrasse de la nécessité de jongler avec les formats de paquets exotiques (TSSOP), nous économise de l'argent (pas besoin de si4021) et surtout, nous évite de concevoir et construire une antenne qui fonctionnera certainement de façon sous-optimale (je l'ai dit, les antennes utilisent une technologie appelée « magie noire » difficile à maîtriser par les non-initiés comme moi).

La télécommande se démonte très facilement puisqu'il suffit de retirer une vis, cachée derrière un autocollant « QC ». Dessouder le HS1527 sera plus ou moins facile en fonction de l'équipement à votre disposition. Je vous recommande la lecture d'un précédent article sur le sujet dans le numéro 18, tout en gardant à l'esprit qu'ici, ce qui nous intéresse n'est pas le composant, mais le circuit imprimé. Peu importe donc qu'il soit éventuellement détruit au passage, l'important étant que les pistes du circuit restent intactes.

Ceci fait, bien entendu, les boutons de la télécommande ne fonctionneront plus, mais nous pouvons utiliser l'emplacement où se trouvait le HS1527, et en particulier les broches de masse et TXD, pour y souder des câbles. On reliera ensuite les masses ensemble et le câble TXD à la broche D4 de la carte Wemos.

En dehors de l'élimination du code de gestion SPI et du si4021 devenu inutile, le croquis ne change pas. Nous n'avons plus besoin de configurer quoi que ce soit et il nous suffit d'utiliser la bibliothèque *rc-switch* avec la broche D4.

Au final, il s'avère que cette version du montage est bien plus stable et efficace que sa déclinaison si4021. Nul doute que la réalisation d'une antenne avec des connaissances limitées en la matière n'est pas étrangère au phénomène. Le circuit pourra reprendre place dans le boîtier de la télécommande qui sera alors découpé pour permettre la connexion. Ceci est presque indispensable puisque la pile A23 12V est toujours nécessaire et que le boîtier assure son maintien.

Le seul élément négatif de cette approche est, bien entendu, le sacrifice d'une télécommande. Dans mon cas avec 6 prises utilisées (pour l'instant) sur les 9 à ma disposition, j'ai réceptionné 3 télécommandes, mais seules deux sont effectivement utilisées. Même si je venais à ajouter des luminaires ou des décorations lumineuses, je n'aurai toujours besoin que de deux télécommandes (une à l'entrée de la pièce, l'autre en balade). Mais en utilisant un seul lot de 3 prises et une unique télécommande, le plus judicieux n'est sans doute pas de la sacrifier pour l'automatisation.

5. RENDRE CECI ACCESSIBLE DEPUIS LE NAVIGATEUR

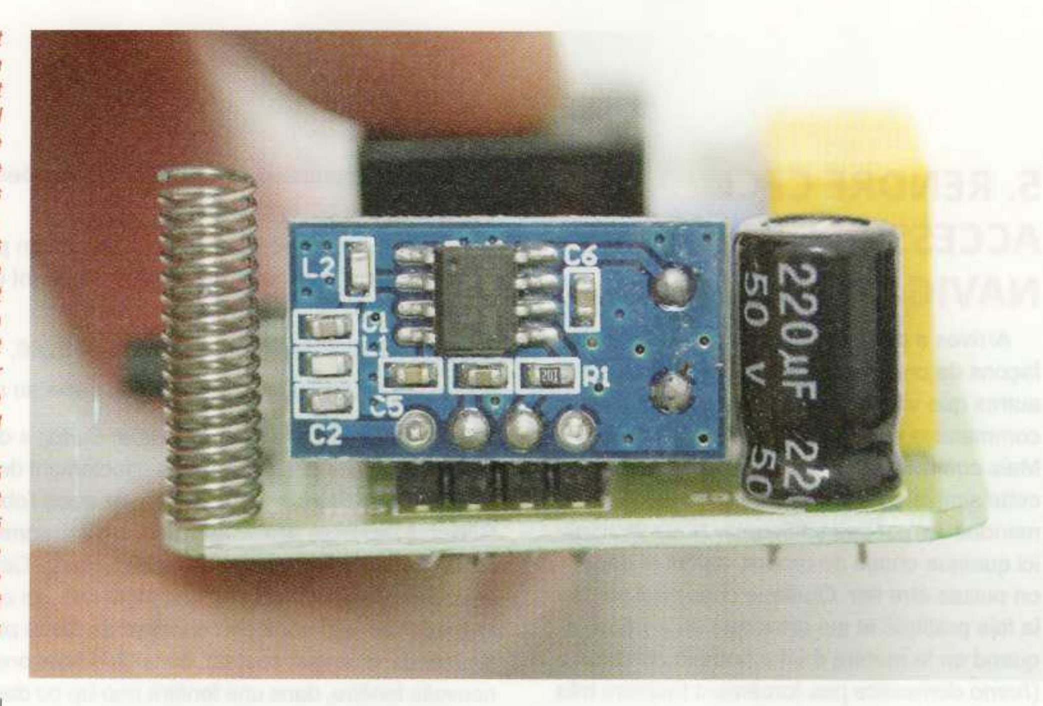
Arrivés à ce stade, nous avons donc deux façons de prendre le contrôle de l'installation, autres que via l'utilisation classique des télécommandes avec nos petits doigts musclés. Mais comme précisé en début d'article, la recette simple du « croquis qui joue les télécommandes » n'est pas suffisante. Nous voulons ici quelque chose de propre, de fini et dont on puisse être fier. Quelque chose qui soit à la fois pratique et qui provoque un « waouh » quand on le montre à un « humain commun » (*homo domestica* pas forcément toujours très *sapiens*), ce qui n'est généralement pas le cas quand on affiche un code ou qu'on joue avec une ligne de commandes.

Diverses approches peuvent être envisagées, mais celle que j'ai choisie est la suivante :

- utiliser une extension Chrome/Chromium pour ajouter un bouton dans la barre d'outils, appelant une page web spécifique,
- fournir cette page web depuis l'ESP8266,
- et donc, forcément, connecter la carte au réseau Wifi.

L'extension utilisée sera « *Custom Button* » de Rubén Martínez et celle-ci est installable directement depuis le *Chrome Web Store*, mais vous pouvez aussi l'obtenir via GitHub à l'adresse **https://github.com/rubenmv/chrome-extension-custom-button** (licence GPLv3). Celle-ci permet de tout simplement spécifier une page web qui sera accédée lors d'un clic sur l'icône personnalisable. Cette page peut s'ouvrir dans l'onglet courant, dans un nouvel onglet, dans une nouvelle fenêtre, dans une fenêtre pop-up ou dans une fenêtre miniature apparaissant sous le bouton (choix à mon sens le plus ergonomique). La fenêtre disparaît d'elle-même dès lors qu'on clique ailleurs ou que le navigateur perd le focus.

Les prises contiennent ce qui semble être un module tiers et le circuit imprimé sur lequel il repose est marqué « BH9938-L ». Cette inscription conduit, lors d'une recherche web, à un type de prises commandées de marque Mercury, avec un circuit identique, mais un module différent. Cet autre module repose sur le composant RF83 de HopeRF pour lequel il existe une documentation, contrairement au A1715 de Steel Tower présent dans nos prises. Ceci peut constituer une piste intéressante pour qui voudrait explorer davantage ces récepteurs...



Tout ce que nous avons à faire donc, est de connecter notre ESP8266 en Wifi, y faire fonctionner un serveur Web (HTTP), composer une page qui présentera des boutons qui réagiront en envoyant les messages à destination des prises pour éteindre toutes les prises d'un mode et allumer celles d'un autre.

C'est à ce stade de l'article que je vais devoir vous faire un aveu : je déteste le développement web. HTML, CSS, JavaScript, Ajax et autres JQuerytudes sont, pour moi, un véritable supplice, un mal nécessaire ne m'apportant aucune satisfaction et aucun plaisir. C'est donc dans cet état d'esprit que j'ai dû me pencher sur la question et la solution finalement choisie fera peut-être hérisser les poils aux « développeurs » web aguerris. En effet, dans le but de ne pas changer de page lors d'un clic sur un bouton, la page en question comprend un élément *iframe*. Nous avons donc un `index` pour la racine du site et deux pages, `/confort` et `/travail`, dont le contenu de type `text/plain` est chargé dans l'iframe lors du clic sur chaque bouton présenté.

J'ai souhaité ces boutons un peu plus travaillés que de simples boutons de formulaires et me suis donc dirigé vers un site proposant des exemples tout faits : **https://freshdesignweb.com/css3-buttons/**. On trouve ici toute une collection de boutons forts sympathiques qu'il sera possible d'intégrer relativement rapidement en étudiant le code HTML de la page et la CSS associée. La page présentée par l'ESP8266 sera donc la suivante :

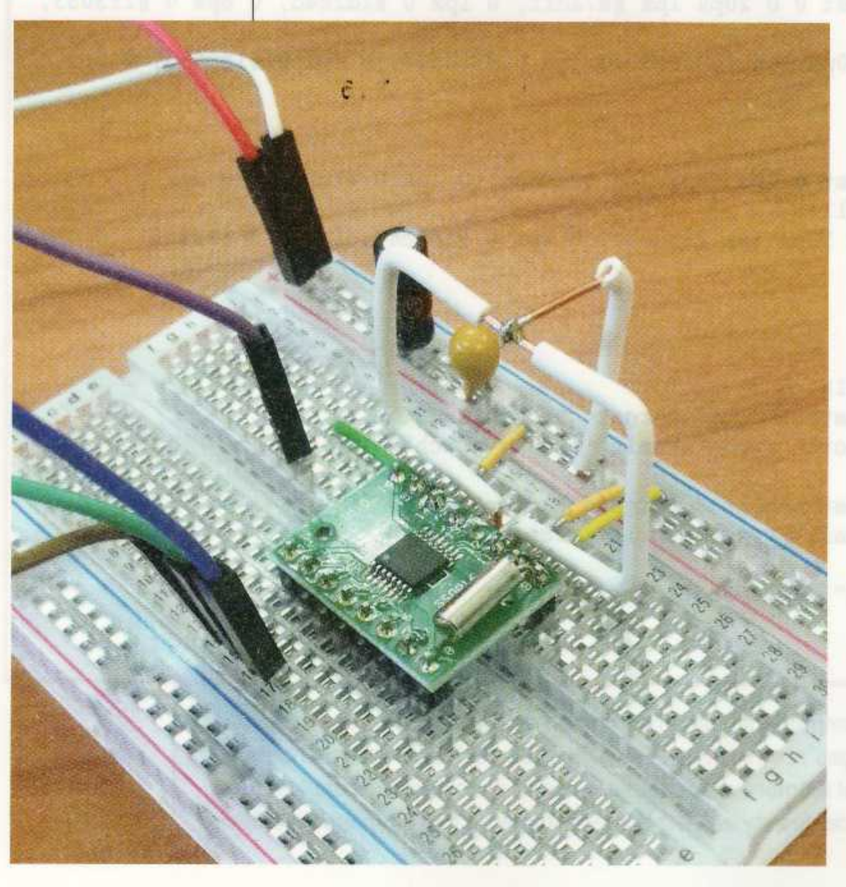
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ESP8266 Contrôle éclairage</title>
<style>
body { font-family: "helvetica neue", helvetica, arial, sans-serif; background: #eee; }
.notif { width: 250px; height: 50px; border: 0; border: none; position: absolute;
margin-top: 10px; }
.punch {
```

```
  outline: none;
  background: #4162a8;
  border-top: 1px solid #38538c;
  border-right: 1px solid #1f2d4d;
  border-bottom: 1px solid #151e33;
  border-left: 1px solid #1f2d4d;
  border-radius: 4px;
  -webkit-box-shadow: inset 0 1px 10px 1px #5c8bee, 0 1px 0 #1d2c4d, 0 6px 0
#1f3053, 0 8px 4px 1px #111111;
  box-shadow: inset 0 1px 10px 1px #5c8bee, 0 1px 0 #1d2c4d, 0 6px 0 #1f3053, 0 8px
4px 1px #111111;
  color: #fff;
  font: bold 20px/1 "helvetica neue", helvetica, arial, sans-serif;
  margin-top: 10px;
  margin-bottom: 10px;
  padding: 10px 0 12px 0;
  text-align: center;
  text-shadow: 0 -1px 1px #1e2d4d;
  width: 150px;
  -webkit-background-clip: padding-box; }
.punch:hover {
  -webkit-box-shadow: inset 0 0 20px 1px #87adff, 0 1px 0 #1d2c4d, 0 6px 0 #1f3053,
0 8px 4px 1px #111111;
  box-shadow: inset 0 0 20px 1px #87adff, 0 1px 0 #1d2c4d, 0 6px 0 #1f3053, 0 8px
4px 1px #111111;
  cursor: pointer; }
.punch:active {
  -webkit-box-shadow: inset 0 1px 10px 1px #5c8bee, 0 1px 0 #1d2c4d, 0 2px 0
#1f3053, 0 4px 3px 0 #111111;
  box-shadow: inset 0 1px 10px 1px #5c8bee, 0 1px 0 #1d2c4d, 0 2px 0 #1f3053, 0 4px
3px 0 #111111; }
</style>
</head>
<body>
<h1>Contrôle éclairage</h1>
<form action="/confort" method="get" target="hiddenFrame">
  <input value="Mode confort" type="submit" class="punch">
</form>
<form action="/travail" method="get" target="hiddenFrame">
  <input value="Mode travail" type="submit" class="punch">
</form>
<iframe name="hiddenFrame" class="notif"></iframe>
</body>
</html>
```

Nous avons là deux formulaires comprenant chacun un bouton de style/classe **punch** provoquant l'accès respectivement aux pages **/confort** et **/travail** avec pour cible l'iframe nommé **hiddenFrame** placé juste dessus. Cet élément est également stylisé avec une CSS et plus précisément la classe **notif**. En l'absence des styles, la page ressemblerait tout simplement à ceci, qui est bien plus intelligible :

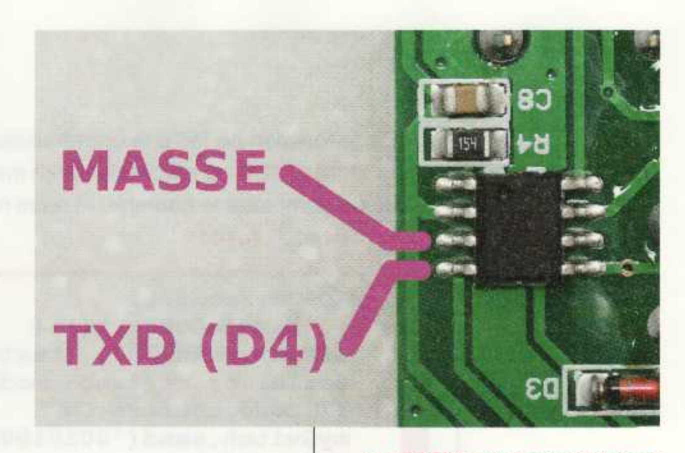
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>ESP8266 Contrôle éclairage</title>
<body>
<h1>Contrôle éclairage</h1>
<form action="/confort" method="get" target="hiddenFrame">
  <input value="Mode confort" type="submit">
</form>
<form action="/travail" method="get" target="hiddenFrame">
  <input value="Mode travail" type="submit">
</form>
<iframe name="hiddenFrame"</iframe>
</body>
</html>
```

Je crois que ceci est précisément ce que je déteste dans HTML/CSS. Ce contraste entre la simplicité de quelque chose qui sera affiché avec une laideur édifiante et la diarrhée syntaxique qui est générée dès lors qu'on tente de faire quelque chose de visuellement plaisant. Passons...



La première approche utilisée pour prendre le contrôle des prises via un montage à base d'ESP8266 repose sur le si4021. Relativement pénible à mettre en œuvre, ce composant présente l'avantage d'être très polyvalent, mais la construction d'une antenne réellement efficace est un obstacle de taille. La portée s'en trouve donc extrêmement limitée, tout comme la fiabilité de l'ensemble.

Le principe ici est de construire la page HTML qui vous satisfait dans un fichier pour en peaufiner les détails avec l'assistance d'un navigateur. Une fois arrivé à un résultat acceptable, il vous suffira de transposer cela dans le croquis de l'ESP8266. Il existe plusieurs façons de faire, car les bibliothèques standards livrées avec le support ESP8266 pour Arduino sont très souples. Vous pouvez gérer les requêtes HTTP vous-même, vous pouvez reposer sur la classe `ESP8266WebServer` ou vous pouvez faire usage du mécanisme SPIFFS permettant d'utiliser une partie de la mémoire Flash SPI de l'ESP8266 comme un espace de stockage (système de fichiers). Notre projet se constituant d'une page unique (ou presque) nous utiliserons la solution qui me parait la plus simple : `ESP8266WebServer` sans SPIFFS (dont il faudra que je vous parle en détail un jour).



Le HS1527 est un circuit intégré pouvant se suffire à lui-même, mais la télécommande n'utilise pas toutes ces fonctionnalités. Ainsi, toute la partie « radio » est gérée par une autre zone du circuit, pour notre plus grand plaisir. En effet, en retirant le HS1527, on peut très facilement y substituer notre ESP8266 à l'aide de deux simples connexions : la masse et la broche recevant le message à émettre.

Pour transformer votre page HTML de façon à ce qu'elle soit utilisable dans votre croquis, il vous suffit d'en fait une variable de type pointeur sur un tableau de `char` :

```
static const char *INDEX_HTML =
"<!DOCTYPE html>\n"
"<html>\n"
"<head>\n"
[...]
;
```

Je vous fais grâce ici de l'ensemble du contenu de cette déclaration puisqu'il s'agit, ni plus ni moins que du code HTML que nous venons de voir. Attention cependant, les guillemets utilisés dans le HTML doivent être précédés d'un `\` pour qu'ils ne soient pas interprétés par le précompilateur comme bornant les chaînes de caractères. Notez également que chaque ligne se termine par un `\n` pour obtenir un saut de ligne (LF) afin que tout le code HTML n'apparaisse pas sur une seule ligne en cas de mise au point ou d'analyse du HTML avec le navigateur.

Nous ajoutons également, dans le croquis, les autres éléments et variables dont nous avons besoin pour la connectivité Wifi et la mise en route du service :

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WebServer.h>
#include <ESP8266mDNS.h>

// SSID du point d'accès
const char* ssid = "monSSIDap";
// mot de passe wifi
const char* password = "MonMotDePasse";

ESP8266WebServer server(80);
```

Notez que nous déclarons ici un objet de type `ESP8266WebServer` et non `WiFiServer` comme dans le numéro 18. Nous comptons créer un serveur HTTP (port 80) pour ne répondre qu'à des requêtes HTTP et, dans ce cas, les bibliothèques du support ESP8266 nous offrent des facilités, dont le fait de ne pas avoir à décoder et interpréter le protocole HTTP nous-mêmes.

Notre serveur web aura cependant besoin de fonctions spécifiques pour répondre à certaines requêtes. Ces fonctions seront ensuite associées à des URL accessibles sur notre serveur et nous commençons par nous occuper de la racine :

```
// racine du site
void handleRoot() {
  server.send(200, "text/html", INDEX_HTML);
  Serial.print("web: accès page racine\n");
}
```

Cette fonction ne fait pas grand-chose si ce n'est renvoyer la page que nous avons stockée dans **INDEX_HTML** et afficher un petit message sur le moniteur série. On voit ici comme il est facile de composer le contenu, et nous nous empressons de faire de même pour les URL **/confort** et **/travail** :

```
// page /confort
void handleConfort() {
  server.send(200, "text/plain", "mode confort activé");
  Serial.print("web: mode confort\n");
  // mode confort ON
  mySwitch.send("001010001010000011011010");
  delay(100);
  // mode travail OFF
  mySwitch.send("001010001010000011010001");
}

// page /travail
void handleTravail() {
  server.send(200, "text/plain", "mode travail activé");
  Serial.print("web: mode travail\n");
  // mode travail ON
  mySwitch.send("001010001010000011011001");
  delay(100);
  // mode confort OFF
  mySwitch.send("001010001010000011010010");
}
```

Ici les réponses sont au format texte brut (non HTML) et sont destinées à s'afficher dans l'iframe. Concernant le pilotage des prises, nous envoyons le message pour allumer les prises associées à un mode puis le message d'extinction pour celles de l'autre mode. Enfin, nous avons une dernière fonction à définir en cas de demande d'une page inexistante. Ce n'est pas absolument critique puisqu'en principe nous avons une utilisation bornée, mais mieux vaut se montrer raisonnable et prévenant :

```
// page 404
void handleNotFound(){
  String message = "File Not Found\n\n";
  message += "URI: ";
  message += server.uri();
  message += "\nMethod: ";
  message += (server.method() == HTTP_GET)?"GET":"POST";
  message += "\nArguments: ";
  message += server.args();
  message += "\n";
  for (uint8_t i=0; i<server.args(); i++){
    message += " " + server.argName(i) + ": " + server.arg(i) + "\n";
  }
  server.send(404, "text/plain", message);
  Serial.println("web: erreur 404");
}
```

Tout le reste de la configuration se passera dans la fonction **setup()**, à commencer par la configuration et la gestion de la connexion Wifi :

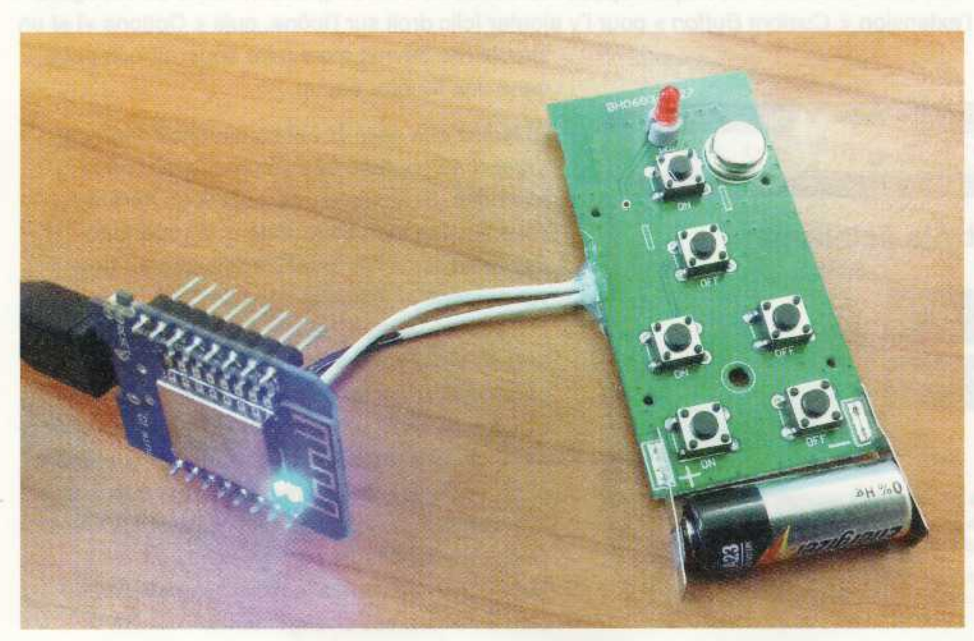
```
// Mode client Wifi
WiFi.mode(WIFI_STA);

// Connexion Wifi
Serial.println("net: Connexion AP...");
WiFi.begin(ssid, password);
while (WiFi.waitForConnectResult() != WL_CONNECTED) {
  // impossible de se connecter au point d'accès
  Serial.println("info: Erreur connexion Wifi !");
  delay(5000);
}

Serial.println("net: WiFi connecté");
Serial.print("net: mon IP = ");
Serial.println(WiFi.localIP());
```

Ceci est très classique et ces lignes de code se retrouvent dans tous les exemples accompagnant le support ESP8266. Comme nous ne comptons pas courir après l'adresse IP du projet à chaque redémarrage, nous faisons usage de mDNS (ou *Multi-cast-DNS*). Ceci nous permettra d'accéder au serveur web avec un nom d'hôte et non une adresse qui peut être variable (DHCP) :

```
// démarrage mDNS-SD
if (!MDNS.begin("esp8266telec")) {
  Serial.println("info: Erreur configuration mDNS!");
} else {
  Serial.println("net: répondeur mDNS démarré");
  // Ajout service disponible ici
  MDNS.addService("http", "tcp", 80); // Announce esp tcp service on port 23
}
```



Avec un peu de patience et de minutie, on pourra souder deux câbles à la place du HS1527 permettant la connexion à l'ESP8266. Celui-ci n'a alors plus qu'à générer une série de pulsations correspondant à l'encodage d'un message binaire pour prendre le contrôle de n'importe quelle prise correspondante. Ceci passe par le sacrifice d'une télécommande, mais le résultat est sans commune mesure avec l'utilisation d'une antenne « bricolo ».

Notre serveur sera ici accessible via `esp8266telec.local` avec n'importe quel système supportant mDNS... ce qui n'est pas le cas de Windows. Une solution très simple pour ajouter ce support consiste à installer les « Services d'impression Bonjour pour Windows » d'Apple. En effet, « Bonjour » n'est autre qu'une implémentation Apple de l'ensemble de protocoles connus sous la désignation Zeroconf et intégrant, entre autres, mDNS. C'en est presque comique, pour faire sous Windows ce que GNU/Linux fait par défaut comme un grand sur une Pi (entre autres), il faut installer un logiciel Apple...

Enfin, nous pouvons configurer les URL et lancer le serveur pour terminer notre fonction `setup()` :

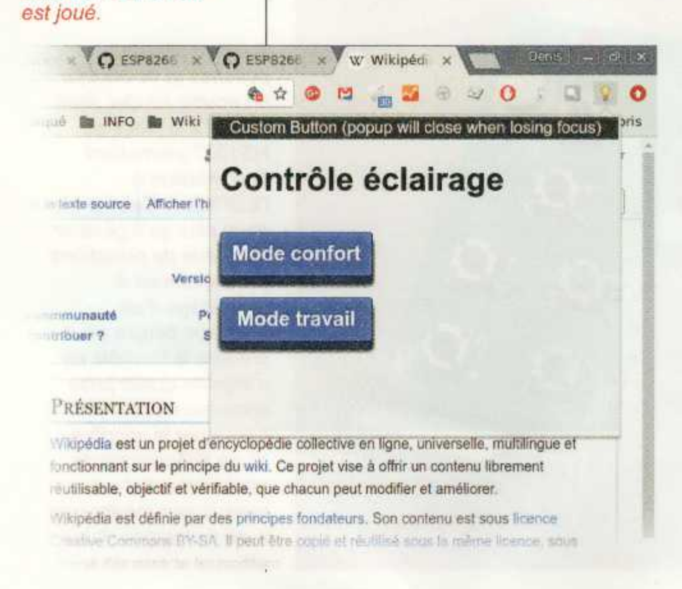
```
// configuration serveur Web
server.on("/", handleRoot);
server.on("/confort", handleConfort);
server.on("/travail", handleTravail);
server.onNotFound(handleNotFound);
server.begin();
Serial.println("web: serveur HTTP démarré");

Serial.println("info: configuration terminée");
Serial.println("info: attente des connexions...");
```

Pour que tout cela fonctionne, il faut encore écrire la fonction `loop()` qui aura pour seule et unique tâche de gérer les connexions à notre serveur web :

```
void loop() {
  server.handleClient();
}
```

Une fois le croquis compilé et enregistré dans la flash de l'ESP8266, la page web devrait être accessible via **http://esp8266telec.local**. On s'empressera alors de configurer l'extension « *Custom Button* » pour l'y ajouter (clic droit sur l'icône, puis « Options ») et un simple clic devrait alors faire apparaître la page dans une fenêtre popup.



Et voilà ! Plus besoin d'attraper la télécommande pour passer à un éclairage tamisé en cas d'envie subite de regarder une petite vidéo. Deux petits clics et le tour est joué.

Notez que celle-ci est surdimensionnée par rapport au contenu très sommaire que nous affichons. Si, comme moi, cela vous dérange, vous pouvez faire un tour dans le répertoire des extensions Chrome/Chromium pour trouver et modifier le fichier de style CSS `popup.css` (`width:400px;` et `height:300px;`). Pour trouver où se trouve ce répertoire, utilisez l'URL chrome://version et regardez à la ligne « Chemin d'accès au profil ». Vous trouverez à cet emplacement un sous-répertoire `Extensions` contenant plusieurs sous-répertoires aux noms abscons, l'un d'entre eux doit contenir l'extension « *Custom Button* », identifiable par son fichier `README.md`. Notez qu'une éventuelle mise à jour de l'extension écrasera vos changements.





Le circuit modifié de la télécommande peut retrouver place dans son boîtier d'origine. Bien sûr, celle-ci n'est plus utilisable dans l'état (via les boutons), mais la pile 12V tient alors solidement en place et nous évitons tous problèmes de faux contacts.

Personnellement, j'utilise principalement Chromium (ou Chrome pour Netflix) et j'ai donc tout naturellement recherché une extension pour ce navigateur. Nul doute cependant que quelque chose d'équivalent doit être disponible pour Firefox, voire pour d'autres navigateurs moins populaires.

6. EXTENSIONS POSSIBLES...

Pour conclure cet article déjà très gros, sinon trop gros aux dires de certaines (au passage si vous ne voyez pas arriver le numéro 25, c'est que l'infographiste en charge de *Hackable* m'aura assassiné #JusticePourLeRédacChef), je préciserai que ce n'est pourtant que la partie émergée de l'iceberg. Il est possible d'aller plus loin, beaucoup plus loin, tant au niveau de l'implémentation que des extensions possibles en termes de fonctionnalités pratiques.

Nous avons déjà accompli pas mal de choses : décoder les messages des télécommandes, trouver le moyen de remettre à zéro la mémoire des prises, découvert comment ajouter le support mDNS dans Windows, éviter d'utiliser un montage complexe pour simuler une télécommande, ajouter un bouton dans le navigateur... et même fait un peu de HTML/CSS qui ressemble à quelque chose.

Mais imaginez à présent tout ce qu'il reste à faire ! En vrac : ajouter une simulation de présence pour décourager les voleurs, changer le mode automatiquement en cas de visite de certains sites (YouTube, Netflix, Xhamster... oups, je veux dire, Dailymotion), associer le changement de mode au lancement d'applications comme des jeux, proposer une page plus adaptée pour les smartphones, permettre un contrôle à distance au travers d'un VPN, offrir la possibilité de configurer de nouveaux modes, connecter le récepteur d'une des prises pour ajouter une fonction d'apprentissage, modifier l'extension pour gérer plusieurs icônes et pages, permettre d'autres protocoles de communication comme MQTT, gérer les prises individuellement et sous forme de modes...

Comme vous le voyez, les idées ne manquent pas et nous aurions de quoi remplir tout un magazine, sinon plusieurs. Mais ceci est maintenant votre terrain de jeu qui, je l'espère, vous amusera tout autant qu'il m'a amusé pour la confection de cet article. Excusez-moi, mais je dois vous laisser, j'ai des lumières à contrôler... jour, nuit, jour... nuit... jour, nuit, jour... DB

ROBOTIQUE ET ÉLECTRONS : MESURER UNE TENSION AVEC LE RPI

Laura Bécognée



Trivial ? Sans intérêt ? Et si je vous disais qu'au final, cet exercice de style vous permettra de connaître avec finesse la consommation d'un moteur électrique pour ajuster son comportement de façon dynamique ?

En théorie, un convertisseur analogique/numérique (CAN) devrait suffire, il a été prévu pour mesurer des tensions et les transformer en données numériques.

Toutefois mesurer une tension, c'est bien joli, mais sans connaître l'intensité électrique qui y est associée, on ne va pas très loin. Nous découvrirons donc, dans un article suivant, comment mesurer celle-ci, avec un circuit astucieux : le *Current Sense Amplifier* (CSA) qui transforme un courant en tension.

Afin de calibrer notre circuit et le Pi qui procèdera à la lecture des résultats, nous commencerons par mesurer la consommation électrique de quelques LED. Mais il nous faudra ensuite passer aux moteurs électriques, et intégrer à notre circuit la gestion de leur alimentation par PWM, grâce à un driver spécifique : *un pont en H*. Ce sera l'objet d'un troisième article.

1. LE CONVERTISSEUR ANALOGIQUE-NUMÉRIQUE

Ce circuit intégré est donc la clé de notre réussite. Grâce à une recherche patiente et à des conseils de pro (merci Whygee !), j'ai réussi à en trouver un qui répond exactement à nos besoins : l'ADC121S101 de Texas Instruments. Derrière ce nom barbare se cache un ADC (*Analog-to-Digital Converter*, CAN en français) capable de fournir des données sur 12 bits, ce qui est largement assez précis pour nos besoins, et supportant des tensions de lecture comprises entre 0 et 5 Volts. Il fonctionne sur le principe du SAR (*Successive Approximation Register*), dont nous aborderons le principe plus loin. Sa vitesse d'échantillonnage optimale tourne entre 500Ksps et 1 Msps (sps : *samples per second*), ce qui est très suffisant pour nos besoins. Pour des applications nécessitant une plus grande précision, par exemple pour faire de la radio, une technologie plus rapide encore est recommandée. Flash est souvent cité. La technologie Sigma delta, elle, s'adresse à des applications nécessitant une plus grande bande passante.

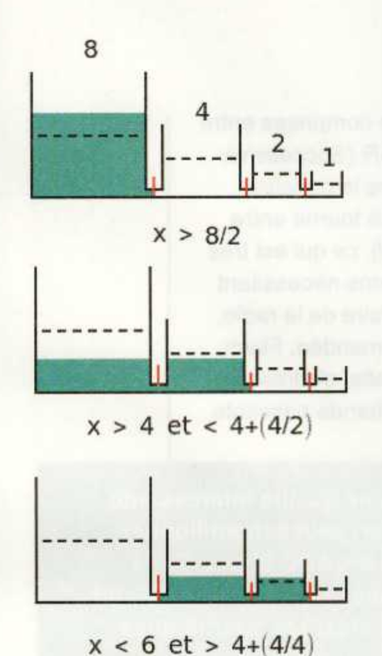
Note : les ADC de type SAR ont de plus une qualité intéressante : pas besoin d'une horloge ultra stable et on peut échantillonner les valeurs quand on le désire, alors que les ADC Sigma-Delta demandent un échantillonnage très régulier. Le SAR réduit la complexité du matériel et du logiciel pour les mesures sporadiques.

1.1 Dessine-moi un pifomètre de précision

Disons que je doive deviner un nombre entier réel positif compris entre 1 et 4. Je vais commencer par être astucieuse et proposer 2. Si vous me dites que c'est plus, alors je proposerai 3 (2+la moitié de 2). C'est encore plus ? C'est donc forcément 4. Il m'a suffi de diviser le chiffre que j'avais proposé par deux et de faire la somme avec le précédent en m'orientant dans la bonne direction en fonction des indications (+ ou -). C'est ce qu'on appelle une recherche dichotomique, et c'est le fonctionnement théorique d'un ADC à SAR.

Ce composant comprend un assemblage de condensateurs aux dimensions parfaitement calibrées, reliés par des transistors qui font passer le courant entre eux, ce qui permet de répartir leur charge, de division en division, jusqu'à obtenir le résultat le plus précis possible en fonction de la largeur de bits en sortie : N condensateurs pour N bits de résolution.

Une autre métaphore passe par l'analogie hydraulique : disons que je doive deviner une quantité de liquide. Pour ce faire, je dispose de quatre seaux de plus en plus petits, reliés entre eux par le fond. Je peux faire passer l'eau de l'un à l'autre avec de petites vannes. Je commence par remplir mon premier seau, de 4 litres, et je constate qu'il est rempli plus qu'à moitié. Il contient donc plus de 2 litres. Je marque 1 sur mon tableau (0 sinon) et j'ouvre ensuite le robinet pour que l'eau se répartisse de façon égale entre mon premier seau

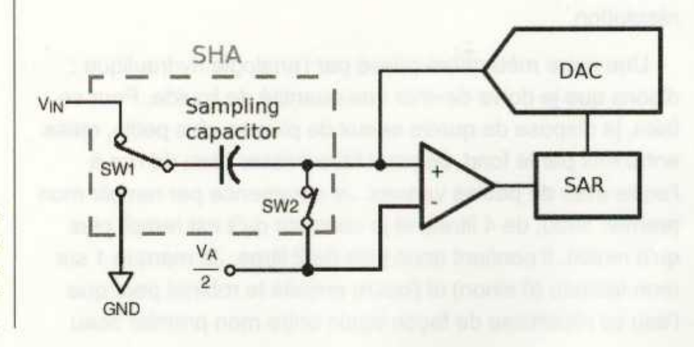


L'ADC selon la métaphore hydraulique.

et le second, deux fois plus petit. Si l'eau monte moins qu'à moitié dans celui-ci, je saurai que mon liquide mesure plus de 2 litres, mais moins de 2 + (2/2) = 3 litres. Je procède ainsi avec les seaux suivants et j'aurai une approximation relativement précise de ma quantité de liquide, sous forme de nombre binaire traduisible ensuite en nombre décimal.

Bien sûr, d'autres éléments gravitent autour de ce SAR afin de le rendre utilisable. Il ne suffit pas de balancer l'eau au hasard en ouvrant et fermant les vannes au petit bonheur la chance, il faut le faire en suivant un timing et un ordre bien précis. C'est pourquoi, on trouve en renfort du SAR d'autres circuits primordiaux : une horloge et un SHA (*Sample-and-Hold Amplifier*) pour hacher, amplifier le signal en entrée et le transmettre au bon moment au SAR, et un comparateur accompagné d'un DAC, qui retransforment le signal numérique en donnée analogique et le comparent au voltage d'entrée jusqu'à ce que les deux soient identiques et le travail de conversion achevé.

Le schéma ci-dessus montre ce que donne ce fonctionnement théorique appliqué à notre circuit intégré. Il est inspiré de la figure 16 du datasheet de la puce. On peut apercevoir ce que sera le SHA, entre les pointillés. Les switchs SW1 et SW2 peuvent être inversés pour cesser la comparaison et garder le résultat en mémoire, afin de le transmettre au monde extérieur : c'est le mode Hold. Va/2 (le voltage d'alimentation du convertisseur divisé par 2) sera alors utilisé comme référence par le comparateur. La puce ne génère pas son horloge, mais il y a une astuce : nous lui fournissons un signal d'horloge sur l'entrée SCLK (*Signal Clock*) qui sert aussi à la communication série avec le RPi.



Un résumé du contenu de notre convertisseur ADC.

1.2 Création d'un circuit de test

Maintenant que nous savons qui fait quoi et comment, nous pouvons entamer sereinement la phase de montage, afin de commencer à utiliser le convertisseur. Nous commencerons par tester une simple LED rouge, qui sera alimentée en 1.8 Volt par une alimentation dédiée, et découvrirons comment récupérer cette valeur sur le RPI, en utilisant le bus SPI, dont nous parlerons bientôt plus en détail.

La création du circuit de test en soi ne pose pas trop de problèmes, à l'exception du montage du CAN que je n'ai pu trouver qu'en SOT23 hélas. N'ayant pas encore reçu les cartes adaptatives SOT23 vers DIP que j'avais commandé, je me suis résolue à les souder directement sur des fiches DIP, ce qui ne fut pas une mince affaire. Ce problème devrait être réglé pour le prochain article.

L'utilisation du voltmètre et de l'Ohmmètre à chaque étape du circuit est une bonne idée, car on n'est jamais à l'abri d'une faute d'étourderie, même si les tensions en jeu restent faibles. Afin de limiter les possibles perturbations de tension dues aux alimentations (lesquelles sont reliées au secteur et peuvent être imparfaitement protégées en interne), il est plus sage d'ajouter entre les broches de chacune un petit condensateur de découplage, qui va absorber les fluctuations et permettre à notre convertisseur de travailler avec moins de bruit.

Ceci étant fait, voici le résultat en photo chez moi. À noter que

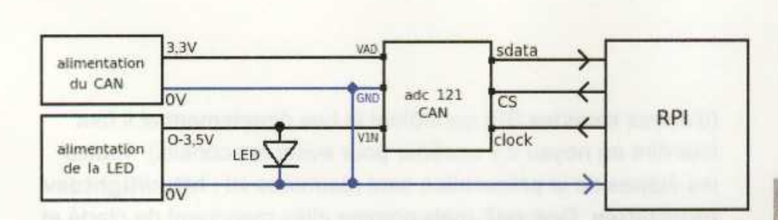


Schéma du montage final.

j'utilise une alimentation externe pour la puce, même si je sais que beaucoup utilisent directement le 3,3 V du RPI. À chacun de voir. Ma seconde alimentation, celle qui alimente la LED, alimentera plus tard les moteurs et il me faudra bien me résoudre à me passer du RPI pour cette partie. De plus, celle que j'ai choisie est dotée d'un potentiomètre multitours et d'un petit afficheur 7 segments qui me permettent de calibrer la tension. C'est confortable et il est ainsi très facile de vérifier la cohérence du résultat avec celui de notre RPI et de son bus SPI, que nous allons maintenant pouvoir commencer à utiliser.

Notre montage prêt pour un premier test.

Note : parmi les broches du RPI, on trouve MISO et MOSI. MISO signifie Master In, Slave Out (la broche du Pi qui lit/reçoit des données série). MOSI : Master Out, Slave In (la broche du Pi qui émet des données série).

2. ET MAINTENANT, LISONS

2.1 Connexion du circuit et du RPI

Pour franchir cette étape avec brio, je vous recommande le site **https://fr.pinout.xyz/pinout/spi**, qui clarifie toutes les broches et regorge d'explications. Vous devez connecter :

- CS sur la broche SPI0 CE0 : Chip Select, signal habituellement actif lorsqu'il est à 0V. CE signifie quant à lui Chip Enable.
- Clock sur la broche SPI0 SCLK.
- Data sur la broche SPI0 MISO.

Et enfin, n'oubliez pas de repiquer le GND de votre plaque d'essai sur la masse voisine de SPI0 SCLK. Plus les masses sont reliées entre elles, plus faibles seront les perturbations sur le circuit. Comme dirait mon mentor Whygee, *le 0V est sacré et universel.* N'hésitez donc pas à le partager entre tous.

Prenez garde également à ne pas vous mélanger les pinceaux entre SPI1 et SPI0. En informatique, on commence à compter à partir de 0. SPI0 correspond donc bien au premier bus et SPI1 au second.

2.2 Mais en fait, c'est quoi le bus SPI ?

Les broches du port GPIO du Raspberry Pi peuvent être utilisées de diverses façons : elles permettent entre autres de relier un afficheur parallèle (avec le mode DPI : *Display Parallel Interface*). On peut aussi en faire un bus I2C (*Inter-Integrated Circuit*) ou encore un bus SPI (*Serial Pheripheral Interface*). C'est ce dernier bus, sélectionné pour sa plus grande simplicité d'implémentation comparé à l'I2C, que nous utiliserons pour ce projet. Attention, car le convertisseur analogique/numérique qui a été choisi (l'ADC121S101), l'a été en fonction de ce protocole. Un convertisseur I2C ne donnera rien de bon si nous tentons de l'utiliser à sa place en suivant le protocole SPI, et ce malgré le fait qu'il s'agit dans les deux cas d'un protocole série.

2.3 Spidev, le portier du SPI

Spidev est un driver SPI pour le noyau Linux, fonctionnant donc également sous d'autres ordinateurs que le Raspberry Pi et accompagné d'un module pouvant être importé facilement dans du code, qu'il s'agisse de C, de Python ou même de Shell. Étant plus à l'aise avec le Python, c'est ce langage que j'ai choisi pour la suite, bien que les principes généraux dégagés restent les mêmes et soient adaptables sans souci.

2.4 Préalables

Le driver Spidev doit tout d'abord être installé puis activé, car il utilise les fonctionnalités du noyau Linux (d'autres librairies SPI contrôlent le bus directement et il faut interdire au noyau d'y accéder pour éviter les conflits). Toutes les étapes de la préparation sont résumées ici : **http://tightdev.net/SpiDev_Doc.pdf**, mais comme elles manquent de clarté et contiennent une ou deux coquilles, je me suis permis de vous les réexpliquer ici.

Il vous faut d'abord activer le bus SPI dans raspi-conf ou directement dans le fichier `/boot/config.txt` de votre Raspberry Pi (`dtparam=spi=on` devrait être dé-commenté). Pour travailler en Python, il vous faudra bien sûr installer python-dev (python2.7-dev pour être plus précis) et python-setuptools, après une petite mise à jour globale qui ne fait jamais de mal.

Vérifiez également que le SPI n'est pas blacklisté dans `/etc/modprobe.d/raspi-blacklist.conf`. Si ce n'est pas déjà le cas, mettez dans ce fichier un croisillon (#) au début la ligne `spi-bcm2708`. Et après avoir validé toutes ces étapes, pensez à redémarrer afin que le noyau prenne en compte toutes les modifications.

Enfin, vous aurez besoin du module Python SpiDev (à ne pas confondre avec le driver ci-dessus). Il vous faudra le compiler, ce qui prendra un temps record avec les commandes ci-dessous :

```
git clone https://github.com/doceme/py-spidev
cd py-spidev
sudo python setup.py install
cd ..
rm -rf py-spidev
```

Pour vérifier que le driver est bien opérationnel, tapez `/dev/spidev` dans la ligne de commandes du shell et laissez l'autocomplétion faire son travail. Vous devriez trouver `/dev/spidev0.0` et `/dev/spidev0.1`. Comme nous l'avons vu plutôt, il y a deux bus SPI sur le RPI (nous utiliserons le premier : spidev0.0). Vous pouvez faire un test plus global avec le fichier spidev_test.c, dont je mets le lien en fin d'article, mais ce test n'est pas indispensable en soi.

2.5 Utiliser le module SPI dans un programme python

Pour utiliser le module SpiDev, après bien sûr l'avoir inclus au début du fichier avec `import spidev`, il vous faudra créer un objet SPI. Si vous parcourez la documentation, vous verrez qu'il existe quelques fonctions bien sympas à l'intérieur de ce module, permettant d'utiliser le SPI dans un éventail de cas

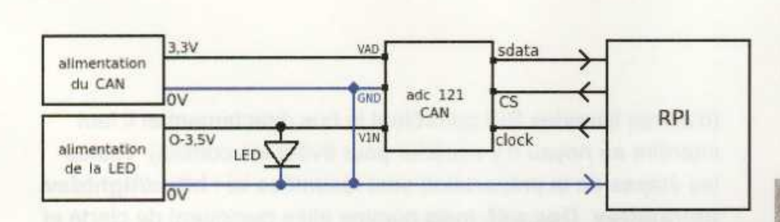


Schéma du montage final.

j'utilise une alimentation externe pour la puce, même si je sais que beaucoup utilisent directement le 3,3 V du RPI. À chacun de voir. Ma seconde alimentation, celle qui alimente la LED, alimentera plus tard les moteurs et il me faudra bien me résoudre à me passer du RPI pour cette partie. De plus, celle que j'ai choisie est dotée d'un potentiomètre multitours et d'un petit afficheur 7 segments qui me permettent de calibrer la tension. C'est confortable et il est ainsi très facile de vérifier la cohérence du résultat avec celui de notre RPI et de son bus SPI, que nous allons maintenant pouvoir commencer à utiliser.

Notre montage prêt pour un premier test.

Note : parmi les broches du RPI, on trouve MISO et MOSI. MISO signifie Master In, Slave Out (la broche du Pi qui lit/reçoit des données série). MOSI : Master Out, Slave In (la broche du Pi qui émet des données série).

2. ET MAINTENANT, LISONS

2.1 Connexion du circuit et du RPI

Pour franchir cette étape avec brio, je vous recommande le site **https://fr.pinout.xyz/pinout/spi**, qui clarifie toutes les broches et regorge d'explications. Vous devez connecter :

- CS sur la broche SPI0 CE0 : Chip Select, signal habituellement actif lorsqu'il est à 0V. CE signifie quant à lui Chip Enable.
- Clock sur la broche SPI0 SCLK.
- Data sur la broche SPI0 MISO.

Et enfin, n'oubliez pas de repiquer le GND de votre plaque d'essai sur la masse voisine de SPI0 SCLK. Plus les masses sont reliées entre elles, plus faibles seront les perturbations sur le circuit. Comme dirait mon mentor Whygee, *le 0V est sacré et universel*. N'hésitez donc pas à le partager entre tous.

Prenez garde également à ne pas vous mélanger les pinceaux entre SPI1 et SPI0. En informatique, on commence à compter à partir de 0. SPI0 correspond donc bien au premier bus et SPI1 au second.

2.2 Mais en fait, c'est quoi le bus SPI ?

Les broches du port GPIO du Raspberry Pi peuvent être utilisées de diverses façons : elles permettent entre autres de relier un afficheur parallèle (avec le mode DPI : *Display Parallel Interface*). On peut aussi en faire un bus I2C (*Inter-Integrated Circuit*) ou encore un bus SPI (*Serial Pheripheral Interface*). C'est ce dernier bus, sélectionné pour sa plus grande simplicité d'implémentation comparé à l'I2C, que nous utiliserons pour ce projet. Attention, car le convertisseur analogique/numérique qui a été choisi (l'ADC121S101), l'a été en fonction de ce protocole. Un convertisseur I2C ne donnera rien de bon si nous tentons de l'utiliser à sa place en suivant le protocole SPI, et ce malgré le fait qu'il s'agit dans les deux cas d'un protocole série.

2.3 Spidev, le portier du SPI

Spidev est un driver SPI pour le noyau Linux, fonctionnant donc également sous d'autres ordinateurs que le Raspberry Pi et accompagné d'un module pouvant être importé facilement dans du code, qu'il s'agisse de C, de Python ou même de Shell. Étant plus à l'aise avec le Python, c'est ce langage que j'ai choisi pour la suite, bien que les principes généraux dégagés restent les mêmes et soient adaptables sans souci.

2.4 Préalables

Le driver Spidev doit tout d'abord être installé puis activé, car il utilise les fonctionnalités du noyau Linux (d'autres librairies SPI contrôlent le bus directement et il faut interdire au noyau d'y accéder pour éviter les conflits). Toutes les étapes de la préparation sont résumées ici : **http://tightdev.net/SpiDev_Doc.pdf**, mais comme elles manquent de clarté et contiennent une ou deux coquilles, je me suis permis de vous les réexpliquer ici.

Il vous faut d'abord activer le bus SPI dans raspi-conf ou directement dans le fichier `/boot/config.txt` de votre Raspberry Pi (`dtparam=spi=on` devrait être dé-commenté). Pour travailler en Python, il vous faudra bien sûr installer python-dev (python2.7-dev pour être plus précis) et python-setuptools, après une petite mise à jour globale qui ne fait jamais de mal.

Vérifiez également que le SPI n'est pas blacklisté dans `/etc/modprobe.d/raspi-blacklist.conf`. Si ce n'est pas déjà le cas, mettez dans ce fichier un croisillon (#) au début la ligne `spi-bcm2708`. Et après avoir validé toutes ces étapes, pensez à redémarrer afin que le noyau prenne en compte toutes les modifications.

Enfin, vous aurez besoin du module Python SpiDev (à ne pas confondre avec le driver ci-dessus). Il vous faudra le compiler, ce qui prendra un temps record avec les commandes ci-dessous :

```
git clone https://github.com/doceme/py-spidev
cd py-spidev
sudo python setup.py install
cd ..
rm -rf py-spidev
```

Pour vérifier que le driver est bien opérationnel, tapez `/dev/spidev` dans la ligne de commandes du shell et laissez l'autocomplétion faire son travail. Vous devriez trouver `/dev/spidev0.0` et `/dev/spidev0.1`. Comme nous l'avons vu plutôt, il y a deux bus SPI sur le RPI (nous utiliserons le premier : spidev0.0). Vous pouvez faire un test plus global avec le fichier spidev_test.c, dont je mets le lien en fin d'article, mais ce test n'est pas indispensable en soi.

2.5 Utiliser le module SPI dans un programme python

Pour utiliser le module SpiDev, après bien sûr l'avoir inclus au début du fichier avec `import spidev`, il vous faudra créer un objet SPI. Si vous parcourez la documentation, vous verrez qu'il existe quelques fonctions bien sympas à l'intérieur de ce module, permettant d'utiliser le SPI dans un éventail de cas

assez confortables. Dans notre cas, **open()**, **xfer2()** et **spi.close()** suffiront. **open()** configurera le port SPI pour nous permettre de l'utiliser, **xfer2()** commandera l'horloge et Chip Select et recevra les informations en provenance de la broche data et **spi.close()** fermera la connexion proprement. À nous ensuite de récupérer ces données et de travailler convenablement avant de les servir au reste du code, ou à un bon vieux **print** pour commencer.

Voilà déjà le code qu'il nous faut pour franchir toutes ces étapes préliminaires :

```
#!/usr/bin/python
import spidev

# creation de l'objet SPI, ouverture et transfert
spi = spidev.SpiDev()
spi.open(0,0)
data = spi.xfer2([0,0], 8000000, 0, 8)
```

Le document trouvable à cette url : **https://pypi.python.org/pypi/spidev**, moins succinct que la documentation officielle, nous apprend que xfer2 prend 4 paramètres en entrée : **[list of values]**, **speed_hz**, **delay_usec** et **bits_per_word**. Nous pouvons déjà paramétrer **[list of values]** : nous voulons récupérer les deux premiers résultats (cf. explication plus bas pour comprendre pourquoi). **Delay_usec** sera laissé à 0 et nous hacherons le résultat en octets : nous inscrirons donc 8 bits pour le paramètre **bits per word**.

Mais il manque la fréquence de l'horloge. Comment la connaître ?

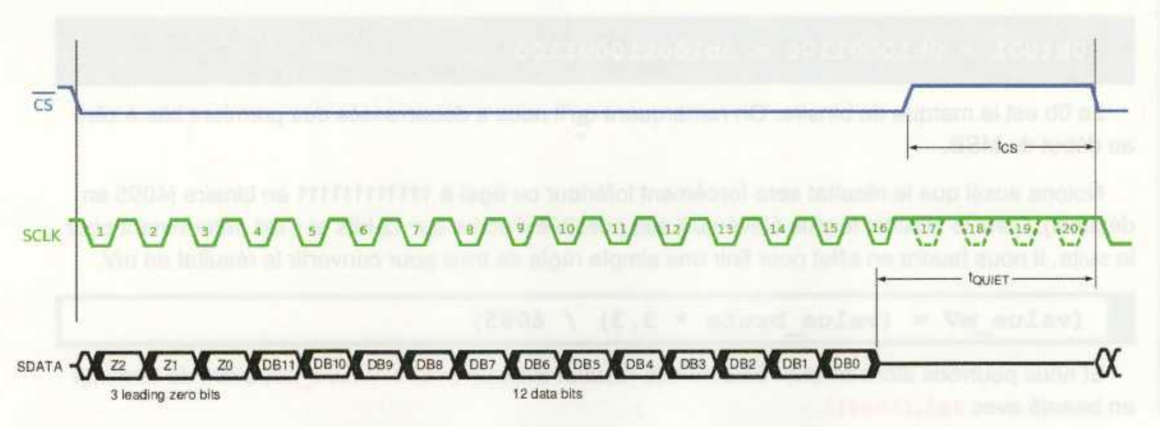
2.6 Le chronogramme

Pour ça, il nous faut retrousser nos manches et explorer les chronogrammes du datasheet. J'ai choisi de me focaliser sur celui de la page 8, que j'ai colorié, simplifié et étendu afin de vous aider à en tirer la substantifique moelle.

Nous pouvons y voir le comportement des broches CS (en bleu), de SCLK (en vert) et de DATA (en noir) les unes par rapport aux autres.

Nous savons déjà que la puce tourne entre 500 Ksps et 1 Msps. C'est-à-dire entre 500 000 et 1 million d'échantillons par seconde. Un échantillon correspond à l'ensemble des données transmises. Ici, le chronogramme nous indique que 16 bits sont transmis en tout : 3 bits laissés à zéro (+ 1 implicite, donc 4 en tout) plus 12 bits de données. 12+4 = 16. Il faut au minimum 500 000 échantillons par seconde et 16 coups d'horloge par échantillon : 500 000 sps × 16 = 8 000 000 Hz. On peut rester sur ce minimum, qui va nous donner un résultat suffisamment précis.

Je dois vous avouer que cette partie-là a été la plus ardue pour moi à comprendre, d'autant plus que la documentation est extrêmement lacunaire sur ce point. J'espère vous faire gagner du temps grâce à ces explications chronographiques.



Le chronogramme débarrassé du superflu.

2.7 Les finitions

À ce stade, si nous rajoutons un print data à la fin, nous obtiendrons un résultat très étrange, de ce style :

```
pi@raspberrypi:~ $ python tension_adc_SPI.py
[8, 246]
```

Mais qu'est-ce que c'est que cette tambouille ?

Deux choses : d'abord, on obtient un résultat en base décimale, et pour autant, notre valeur est bel et bien stockée en binaire. Ça n'aura donc aucune conséquence sur les opérations qui vont suivre, car il utilisera toujours ces dernières. Nous verrons un peu plus bas comment afficher malgré tout le résultat en binaire pour y voir plus clair.

Ensuite, nous avons une liste, contenant deux résultats. En effet, le RPI travaille traditionnellement sur 8 bits. Or notre puce, elle, travaille sur 12. Le SPI a donc haché notre résultat en deux : d'abord les MSB, les bits de poids forts, les plus à gauche de notre résultat, puis les LSB, les bits de poids faibles, les plus à droite, et accessoirement les moins significatifs, mais qui nous permettront d'atteindre une précision salutaire quand nous passerons des LEDs aux moteurs.

Il nous faut concaténer ces bits (les MSB et les LSB), les assembler dans un seul grand nombre binaire de 12 bits, qui sera notre résultat véritable. Voici la méthode permettant d'y parvenir, une simple concaténation.

```
# Oups, la réponse arrive sur deux octets
msb = data[0]
lsb = data[1]
value_brute = (msb << 8) + lsb
```

Pas si terrible, n'est-ce pas ? Nous avons pris les premières entrées de notre liste entre crochets (jusqu'au 8ème bit) et nous les avons collées aux autres.

Si on voulait jouer les curieux, on pourrait ajouter à ce stade cette ligne de code :

```
print (bin(data[0]) + " + " + bin(data[1]) + " = " + bin(value_brute))
```

Ce qui nous renverrait :

```
0b1001 + 0b10001100 = 0b100110001100
```

Le 0b est la marque du binaire. On remarquera qu'il nous a débarrassés des premiers bits à zéro au début du MSB.

Notons aussi que le résultat sera forcément inférieur ou égal à 111111111111 en binaire (4095 en décimal). C'est le résultat le plus élevé qu'il soit possible d'écrire sur 12 bits et c'est déterminant pour la suite. Il nous faudra en effet pour finir une simple règle de trois pour convertir le résultat en mV.

```
(value_mV = (value_brute * 3.3) / 4095)
```

Et nous pourrons alors afficher fièrement le résultat avec un `print value_mV`, avant de terminer en beauté avec `spi.close()`.

Et voilà le code complet :

```
#!/usr/bin/python
import spidev
# creation de l'objet SPI et initialisation
spi = spidev.SpiDev()
spi.open(0,0)
data = spi.xfer2([0,0], 8000000, 0, 8)

# Oups, la réponse arrive sur deux octets
msb = data[0]
lsb = data[1]
value_brute = (msb << 8) + lsb
value_mV = (value_brute * 3.3) / 4095
print (bin(data[0]) + " + " + bin(data[1]) + " = " + bin(value_brute))
print value_mV
spi.close()
```

Et le résultat sera grosso modo :

```
0b1001 + 0b10001111 = 0b100110001111
1.97194139194
```

Lancez le programme plusieurs fois de suite et vous noterez de légères variations. Il y a en effet un léger bruit de X millivolts.

CONCLUSION

Nous savons désormais piloter un ADC depuis le port SPI d'un Raspberry Pi et l'avons transformé en un Voltmètre hyper sophistiqué. C'est déjà un bond de géant, mais il nous reste du travail pour obtenir le banc de test final. Récupérer une tension ne nous apprendra rien, ce qu'il nous faut, c'est un tableau. Et il faut aussi récupérer l'intensité associée afin de déduire la consommation des moteurs. Ce sera l'objet de l'article suivant, dédié à la mesure des variations d'intensité grâce au Raspberry Pi.

J'espère que cet article aura permis d'éclaircir de nombreux points sur l'utilisation du port SPI sur le Raspberry Pi, qui est fort mal documentée sur Internet, ce qui est vraiment dommage, car c'est un moyen simple de rendre son Raspberry Pi encore bien plus utile.

Sur ce, bidouillez bien ! LB

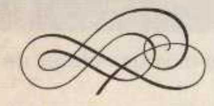
POUR ALLER PLUS LOIN :

- **http://deusyss.developpez.com/tutoriels/RaspberryPi/PythonEtLeGpio/** : un indispensable tutoriel en français, rédigé par Alexandre GALODE, qui m'a servi de fil d'Ariane pour toute la partie SPI.
- **http://sametmax.com** : mes héros en terre pythonienne, bourrés d'humour et qui ont su m'éclairer sur moult questions.
- **http://tightdev.net/SpiDev_Doc.pdf** : la documentation de Spidev, indispensable pour l'utiliser à fond.
- **https://raw.githubusercontent.com/raspberrypi/linux/rpi-3.10.y/Documentation/spi/spidev_test.c** : pour tester le driver spidev.



PILOTEZ LE MYTHIQUE MOS SID AVEC UN ARDUINO POUR JOUER VOS CHIPTUNES

Denis Bodor



Le terme « chiptune », littéralement « mélodie de puce », désigne un genre musical très particulier, se caractérisant par les sonorités typiques issues des puces audio des ordinateurs et consoles vintages, souvent 8 bits, des années 80/90. Il existe bien des façons d'écouter ces musiques mais si, comme moi, on désire l'expérience la plus authentique qui soit, ceci passe généralement par l'utilisation du matériel de l'époque. Mais il existe une autre voie, mitoyenne, consistant à interfacer une puce audio originale avec du matériel moderne comme une carte Arduino.





Voici mon vénérable Commodore 64 en boîtier « breadbin » ou « breadbox » (huche à pain). Cette machine typique des années 80 et du début des années 90 est ce qu'on appelait à l'époque un « ordinateur familial 8 bits ». Vendu à quelques 17 ou 25 millions d'exemplaires (selon les estimations), son succès est en grande partie dû à ses fonctionnalités graphiques et sonores...

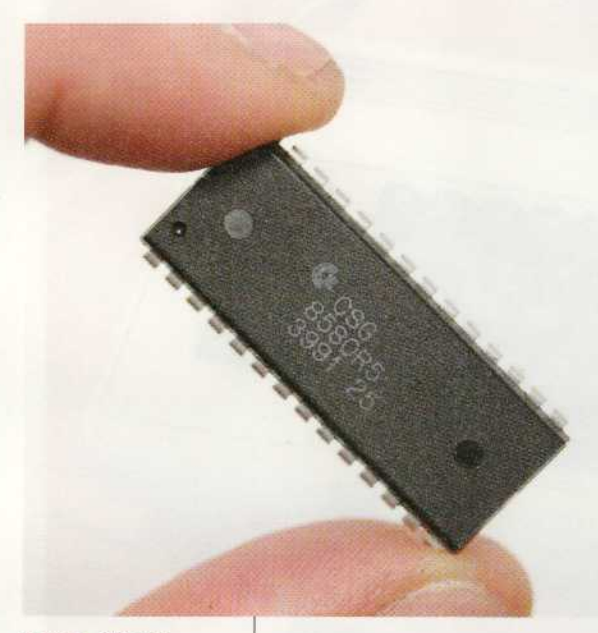
Soyons clairs d'entrée de jeu : ce qui va suivre n'est ni la solution la plus aisée, pratique ou économique, mais sans le moindre doute celle qui est la plus intéressante. Si vous souhaitez simplement écouter des morceaux de musique avec cette sonorité particulière, il existe quantité de lecteurs (*players*) qui feront ce travail à la perfection, aussi bien sur PC que sur la machine ayant servi à créer la musique à l'origine. L'approche que nous allons découvrir est purement technique et ludique, et l'occasion d'apprendre énormément de choses sur des technologies vieilles de quelques 30 ans ou plus.

La nostalgie pour cette sonorité a donné naissance à la notion même de chiptune ou de bitpop et très rapidement, dès la fin des années 90, tout un écosystème s'est développé, porté partiellement par la demoscene. Pour le sujet qui nous intéresse ici, un format de fichier spécifique a été développé afin de pouvoir réécouter ces musiques sans forcément posséder le matériel correspondant. Contrairement à des formats plus « rationalisés » comme MOD et XM provenant de *soundtrackers* initialement développés sur la génération suivante de machines (comme l'Amiga et l'Atari ST), les premiers morceaux de chiptune ne sont pas juste des données. Il s'agit de programmes dont l'exécution génère la musique.

Les fichiers musicaux que l'on trouve sur Internet, comme par exemple sur HVSC (*High Voltage SID Collection* avec quelques 50000 morceaux), n'ont donc rien de comparable avec des fichiers MP3, FLAC, WAV ou même MIDI qui sont simplement lus par un outil. La tâche qui nous incombe ne consistera donc pas simplement à lire des fichiers et jouer leur contenu, nous devrons faire preuve d'une certaine agilité mentale...

1. LE SID

La notion de chiptune n'est pas spécifique à une machine ou une console en particulier, mais plutôt à une époque. Dans de nombreux cas, le « son 8 bits » découle de l'utilisation de technologies propres à cette période. Pour produire du son et de la musique, deux



Voici le SID ! Rien à voir avec le personnage de Toy Story ou de L'âge de glace, ni même avec la pièce de théâtre de Corneille. Le SID, signifiant « Sound Interface Device » est l'œuvre de Bob Yannes et est le nom donné à la puce sonore présente dans les ordinateurs Commodore 64, Commodore 128 et Commodore MAX Machine (alias Ultimax ou VC-10).

approches étaient alors possibles : laisser le processeur faire tout le travail (cas typique de la NES), ou confier la tâche à une puce dédiée qui était un périphérique contrôlé par le processeur. Cette architecture était le précurseur de ce qui deviendra ensuite le standard sur PC : la carte son.

Le composant qui nous intéresse ici est celui qui équipait le Commodore 64 (et 128) : la puce MOS 6581, également appelée SID pour *Sound Interface Device*, le « périphérique d'interface sonore ». Ces composants sont relativement difficiles à trouver aujourd'hui, et lorsqu'on y arrive, ils sont relativement chers. Heureusement, la vie du Commodore 64 n'a pas été un long fleuve tranquille et en 1987 une version modernisée de la machine a été commercialisée. Celle-ci intégrait des déclinaisons améliorées de l'ensemble des composants logiques et donc également du SID, sous la désignation 8580. Cette version est bien plus facile à trouver aujourd'hui.

Sans entrer dans le détail, le 8580 se différencie du 6581 par la technologie utilisée pour sa fabrication : NMOS pour l'ancienne version et HMOS-II pour le 8580. Ceci impacte en particulier son alimentation passant de 12 volts à « seulement » 9 volts. Même si les deux puces produisent un son légèrement différent, c'est précisément cette tension qui est le point critique, car alimenter un 8580 en 12 volts le détruira instantanément. Si vous avez la chance d'avoir sous la main un 6581, je ne peux que vous recommander d'y faire excessivement attention. Ces composants sont extrêmement rares, fragiles et très sensibles. Personnellement, je ne suis pas sûr que je me risquerai à expérimenter avec une telle pièce de collection (je n'ai qu'un seul 6581 et il ne sortira pas du C64 qui lui sert de nid douillet).

Si vous cherchez à acquérir un SID aujourd'hui, vous avez toutes les chances de tomber uniquement sur des 8580R5 (révision 5) produits entre 1986 et 1993. Il vous en coûtera quelques 50€ pour cet unique composant et vous devrez prendre en compte une certaine part de risque. En effet, ces composants ne sont plus fabriqués depuis de nombreuses années et sont pourtant encore très demandés (pour pièce de rechange dans un Commodore 64C, pour modifier/améliorer un C64, ou pour créer un instrument de musique MIDI type SidStation ou MIDIbox SID). On trouve ainsi des composants désignés comme « *new old stock* » (NOS) étant soi-disant des « anciens stocks de pièces jamais utilisées », mais qui sont en réalité souvent des rejets d'usine, des puces en partie défectueuses, voire des composants falsifiés sortant de lignes de production chinoises ressemblant à des SID, mais dont le boîtier contient un autre circuit, voire rien du tout.

Une autre voie pour obtenir un SID consiste à tout simplement acquérir un Commodore 64C nu pour en retirer le composant qui nous intéresse (il n'est pas soudé). Il est possible de tantôt faire de bonnes affaires lorsque la personne ne sait pas vraiment ce qu'elle vend ou s'en moque. Composant seul ou machine complète, peu importe, mais faites preuve de prudence, regardez les évaluations du vendeur, la description, etc. Et méfiez-vous des annonces de type « *je ne l'ai pas testé, je ne sais pas s'il fonctionne* ». Vous pouvez avoir de la chance (j'en ai déjà eu) ou vous pouvez vous faire avoir en tombant sur une personne peu scrupuleuse ayant désossé la machine et prétendant tout ignorer de son état. C'est le jeu...

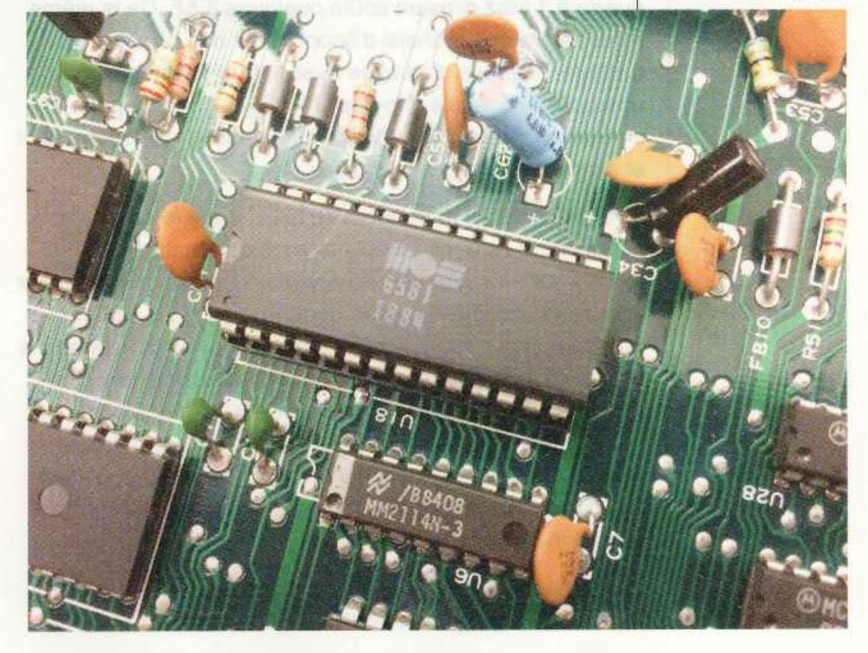
Le SID (6581 ou 8580) possède un certain nombre de caractéristiques qui, pour son époque, étaient très innovantes et faisait de cette simple puce un véritable synthétiseur :

- trois voix indépendantes contrôlées par trois oscillateurs programmables sur 8 octaves (16 à 4000 hz) ;
- quatre formes d'onde pour chaque voix : triangle, dents-de-scie, pulsation (carrée), bruit ;
- un générateur d'enveloppes ADSR (*Attack Decay Sustain Release*) par voix/oscillateur ;
- un filtre multi-mode permettant de combiner les voix pour obtenir des timbres particuliers ;
- trois *ring modulators* pour combiner les fréquences (modulation) ;
- une fonction de synchronisation d'oscillateurs entre les voix permettant de créer des sons plus riches en harmoniques ;
- un générateur de nombres pseudo-aléatoires (bruit) ;
- une entrée audio permettant de mixer le son du SID avec une source externe ;
- deux convertisseurs analogiques-numériques 8 bits, normalement destinés à la connexion de potentiomètres (*paddle* ou souris).

Le SID se présente comme un composant en boîtier PDIP de 28 broches agencées comme ceci :

- 4 broches (1 & 2, 3 & 4) permettant la connexion de condensateurs 22 nF (pour le 8580, et 470pF pour le 6581) destinés au fonctionnement du filtre multi-mode ;
- une broche /RES de réinitialisation (5) ;
- une broche phi (6) pour le signal d'horloge cadençant la puce (1 Mhz) ;
- une broche R/W (7) contrôlant la lecture ou l'écriture dans les registres (la lecture ne sert que pour lire la valeur des convertisseurs analogiques-numériques) ;
- une broche /CS (8) pour adresser/asservir le composant ;
- 5 broches d'adresses (9 à 13) pour accéder aux registres contrôlant le composant ;
- une indispensable masse (14) classiquement en bas à gauche du boîtier ;
- 8 broches D0 à D7 (15 à 22) formant le bus de données de 8 bits pour lire ou écrire les valeurs dans les registres ;
- 2 broches POTY et POTX (23 & 24) pour la connexion de potentiomètres ;
- une broche d'alimentation VCC (25) pour le +5 volts ;
- l'entrée audio EXT IN (26) ;
- la sortie AUDIO OUT (27) ;
- et enfin la tension d'alimentation secondaire VDD de +9 volts pour le 8580 et de +12 volts pour le 6581.

Il existe plusieurs versions du SID. Voici un MOS 6581, une véritable pièce de collection que je ne me risquerai pas à utiliser pour un montage, lui préférant une version plus récente, le 8580, produite par Commodore Semiconductor Group (CSG, anciennement MOS Technology), sensiblement différente, moins coûteuse, moins fragile et surtout plus facile à trouver.



2. PILOTER LE SID AVEC UN CROQUIS ARDUINO

Pour piloter un SID avec, par exemple, une carte Arduino UNO, nous avons plusieurs possibilités en fonction de ce dont la carte doit effectivement gérer elle-même. Le montage qui va être décrit ici a subi plusieurs itérations avant d'arriver à ce que j'estime être quelque chose de « présentable ». Dans la première version, l'ensemble des signaux indispensables était généré par l'Arduino : adresses, données, /CS, reset, R/W et horloge, soit 17 broches en tout. R/W n'est pas utile et peut être directement relié à la masse. En effet, les registres qui nous intéressent sont en écriture seule et ne sont pas lisibles.

Le signal d'horloge lui, est un cas particulier, car il est tout à fait possible de le générer via l'un des timers du microcontrôleur de la carte Arduino, mais la broche utilisée ne peut être arbitrairement choisie (c'est 10 pour l'OC1A du Timer1). Cette solution ne me plaisait pas, même si un code de Asbjørn Brask (alias atbrask), parfaitement fonctionnel l'utilisait. Je me suis donc basé sur sa solution (**https://github.com/atbrask/RealSIDShield**), mais en revoyant totalement le code et sa logique, et en étendant ses fonctionnalités.

Alors que lui utilisait l'Arduino comme source d'horloge, j'ai préféré, tout simplement, faire usage d'un oscillateur à quartz à 1 Mhz m'ayant coûté quelques 3,5€. De la même façon, alors qu'il a choisi d'économiser des broches en partageant le bus d'adresse avec le bus de données via une octuple bascule (flip-flop) 74HC574, j'ai opté pour la connexion directe, économisant au passage un composant et simplifiant le code du croquis.

Enfin dernière différence, de taille, entre le projet d'Asbjørn et celui-ci, toute la logique de fonctionnement repose sur un cadencement par le script Python et non par le croquis Arduino. En effet, comme je l'ai sous-entendu précédemment, les fichiers musicaux que nous souhaitons utiliser ne sont pas de simples morceaux, mais des programmes mêlés de données. Il nous faut donc émuler, en partie, un Commodore 64 pour exécuter ces programmes et en inspecter le fonctionnement. La vitesse à laquelle ces données sont collectées puis envoyées au SID détermine donc la vitesse de lecture.

Pour compliquer encore davantage les choses, cette vitesse est dépendante des choix faits par le créateur du fichier SID. Certains utilisent simplement la vitesse de rafraîchissement vidéo (50 Hz), mais d'autres reposent sur la programmation d'un composant particulier du Commodore 64, le CIA (*Complex Interface Adapter*). Il y a deux CIA dans un C64, servant à lire le clavier, contrôler le port série, lire l'état des ports joystick, etc. Chacun d'eux intègre deux timers programmables et une certaine quantité de morceaux de musique (dont mes préférés) utilisent le timer A du CIA 1 pour cadencer la lecture.

Asbjørn a fait le choix de s'en tenir aux 50 Hz utilisés par seulement une partie des morceaux disponibles et a écrit un script Python envoyant les données à la carte Arduino dès lors que celle-ci signifie qu'elle est prête, 50 fois par seconde. C'est donc le croquis de la carte qui détermine initialement la vitesse de rafraîchissement et de lecture. On retrouve cette limitation dans la source d'inspiration d'Asbjørn, `siddump`, un code en C étrangement similaire avec des commentaires identiques, sans qu'il en soit fait mention nulle part (pas très correct je trouve).

Personnellement, je voulais pouvoir lire les autres morceaux, utilisant le timer du CIA 1 et pour ce faire, il était bien plus simple de donner directement le contrôle de la cadence au script Python. Au final, il ne reste donc plus grand-chose du concept initial si ce n'est dans une partie du fonctionnement du script Python, qui n'est en réalité que la reprise, dans ce langage, du code `siddump` de

Notre montage ne nécessite pas beaucoup de composants, mais énormément de câblage. Les deux points critiques du circuit concernent principalement l'alimentation du SID 8580 (en +5V et en +9V) et la génération d'un signal de 1 Mhz pour cadencer le composant.

Lasse Öörni (alias cadaver) destiné à analyser des fichiers SID (**https://github.com/cadaver/siddump**).

Pour résumer, ce que nous allons mettre en place repose sur l'architecture suivante :

- un script Python simule le fonctionnement partiel d'un Commodore 64 via l'utilisation du module Python *Py65* ;
- le fichier à lire, au format PSID (RSID n'est pas supporté, car trop peu courant) est chargé par le script et placé dans la mémoire émulée ;
- la lecture de l'en tête du fichier détermine la source utilisée pour cadencer la lecture et permet d'afficher diverses informations ;
- la simulation est effectuée tout en surveillant le contenu de la mémoire. À des moments clés de l'exécution, nous lisons la mémoire aux adresses correspondant aux 25 registres du SID et à deux registres du CIA 1 contenant la valeur du timer A sur 16 bits ;
- si le CIA est utilisé, on se sert de la valeur du timer pour cadencer l'exécution et l'envoi des données des registres du SID à la carte Arduino via la liaison série/USB (sinon c'est simplement 1/50 seconde qui doivent s'écouler) ;
- le croquis réceptionne les données et procède à l'écriture des valeurs dans les registres du SID si celles-ci changent.

Notez au passage qu'on ne peut pas réellement parler d'une véritable émulation de C64 puisque nous n'avons que faire de ce que fait exactement le programme dans le fichier SID. Nous estimons simplement que lorsque le processeur simulé arrive sur certaines instructions en langage machine (RTI, RTS, BRK) ou dans certaines zones mémoire (routine d'interruption en ROM), c'est là qu'il faut lire et envoyer l'état des registres au véritable SID. Nous ne faisons rien de plus que l'outil `siddump` si ce n'est ajouter une composante temporelle et envoyer les données via un port série.

3. LE MONTAGE

Finalement, le montage Arduino et son croquis n'auront pas grand-chose à faire si ce n'est de recevoir le contenu destiné aux registres du SID et utiliser les différentes broches du composant pour y écrire. La difficulté ne concerne pas la carte Arduino, mais le SID et en particulier deux points : l'alimentation 9 volts (broche 28)



Il est possible de générer une fréquence de 1 Mhz à partir d'une carte Arduino, mais l'option retenue ici sera l'utilisation d'un oscillateur à quartz. On dispose ainsi, sans effort, d'un composant à 4 pattes (+5V, masse, signal et NC) faisant tout ce travail avec une grande précision et stabilité pour quelques euros.

et le signal d'horloge à 1 Mhz devant être présenté sur sa broche 6.

Les deux points peuvent être réglés de plusieurs manières. Fournir 9 volts au SID peut se faire en utilisant une pile 9 volts (peu recommandé), utiliser une source d'alimentation externe (bloc 9V), utiliser le jack de la carte Arduino en fournissant 9V et donc en connectant Vin au VDD du SID et la broche 5V (régulée) sur VCC, où enfin, utiliser un convertisseur (booster) comme celui présenté dans le numéro 19. Rappelons au passage que ce type de convertisseur de tension très économique existe en deux versions pouvant fournir 12V ou 9V à partir d'un connecteur USB. En démontant le produit pour ne garder que le circuit, on peut donc facilement obtenir la tension pour un 6581 ou un 8580. Et c'est précisément ce que j'ai fait (ayant des convertisseurs en version 12V j'ai, de plus, modifié l'une des résistances comme détaillé dans le numéro 19 pour obtenir 9V).

Le second point concerne la méthode à choisir pour envoyer un signal carré de 1 mégahertz sur la broche *phi* (6) du SID. Normalement, ce signal est fourni par le processeur 6510 du Commodore 64 et deux options nous sont accessibles : utiliser un bout de code pour générer cette fréquence sur la carte Arduino et donc connecter la broche 10 de la carte au SID, ou utiliser un composant supplémentaire, un oscillateur à quartz parfaitement calibré à 1 Mhz. Celui-ci se présente simplement sous la forme un petit boîtier métallique à 4 pattes : alimentation, masse, signal et non-connecté.

Il conviendra cependant de faire attention à votre achat pour cette solution. Un oscillateur à quartz est un montage électronique, un module miniature, à ne pas confondre avec un quartz (à deux pattes) qui n'est littéralement qu'un morceau de quartz nécessitant des composants complémentaires pour fonctionner. Il faudra également faire attention à la tension d'utilisation du composant, qui doit être prévu pour fonctionner en 5 volts. J'ai acquis une poignée de ces composants pour 3,5€ pièce, sur eBay auprès d'un vendeur appelé *ha5ia* (Londres), sous la désignation « *1 MHz Crystal Oscillator osc. 5 V H DIP* ».

Concernant la connexion de la carte Arduino au SID, nous avons la nomenclature suivante :

- bus d'adresse A0 (10) à A4 (13) sur les broches A0 à A4 de l'Arduino ;
- bus de données D0 (15) à D7 (22) sur les broches 2 à 9 ;
- signal /CS (8) sur 11 ;
- reset (5) sur 12 ;
- R/W (7) à la masse.

On connectera également des condensateurs céramiques de 22nF sur les broches 1 et 2, ainsi que sur 3 et 4. Et enfin, nous connecterons la sortie audio à la masse (27) via une résistance de 1 Kohm, ainsi qu'à la broche positive d'un condensateur électronique de 1µF afin de supprimer la tension continue (DC bias) et fournir un signal utilisable avec un équipement audio amplifié. La broche négative du condensateur sera reliée à une prise jack 3,5mm permettant de brancher une enceinte amplifiée (un simple casque ne fonctionnera pas).

Dernier point qui peut être très important concernant ce montage, il est impératif que les tensions d'alimentation (+5V et +9V) du SID soient « propres » et donc exemptes de bruit. Il est donc de bon ton de placer des condensateurs de découplage entre l'alimentation et la masse au plus près du SID. Idéalement, un électrolytique d'une dizaine ou centaine de microfarads pour les basses fréquences et un tantale

goutte de quelque 0,1µF pour les hautes fréquences. Si l'alimentation est relativement propre ou déjà filtrée (par la carte Arduino ou le convertisseur de tension), le tantale devrait suffire (il a suffi dans mes essais).

4. LE CROQUIS ARDUINO

Les données envoyées par le script Python simulant le fonctionnement du processeur du Commodore 64 sont textuelles. Ce n'est pas un choix que j'aurai personnellement fait, mais l'idée ici est justement de ne pas tout redévelopper. Comme nous le verrons dans la conclusion de l'article, mes futurs développements concernant ce projet s'orienteront dans une direction différente ne nécessitant pas d'investir, outre mesure, dans ce code en Python.

La tâche du croquis Arduino consistera donc principalement à réceptionner les données via le port série, les accumuler jusqu'à obtenir un nouveau jeu de valeurs pour les registres, les décoder et enfin, les envoyer au composant.

L'approche consistera, très classiquement, à définir et déclarer les macros et variables utiles puis de créer des fonctions exécutant les tâches les plus simples, pour enfin utiliser le tout dans la boucle principale. Commençons donc par le début :

```
#define D0 2
#define D1 3
#define D2 4
#define D3 5
#define D4 6
#define D5 7
#define D6 8
#define D7 9

#define RESETPIN 12
#define CSPIN 11

// nouvelles valeurs des registres
char newsid[25];
// précédentes valeurs des registres
char oldsid[25];
// drapeau de disponibilité des données
bool dataready = false;
// tampon de lecteur
// 25 registres, 2 caractères par octet
char buffer[50];
// index pour la lecture série
int idx = 0;
```

Les macros **D0** à **D7** correspondent aux broches utilisées pour le bus de données et nous n'avons pas besoin de faire de même pour le bus d'adresse qui, naturellement correspond déjà aux lignes A0 à A4 (les broches pouvant être des entrées analogiques sur une carte UNO). Les tableaux **newsid[]** et **oldsid[]** sont destinés à contenir les valeurs en octets (**char**) des 25 registres accessibles en écriture sur le SID. Nous en avons deux exemplaires afin de pouvoir comparer la dernière valeur utilisée avec la nouvelle et de procéder à l'écriture qu'en cas de changement.

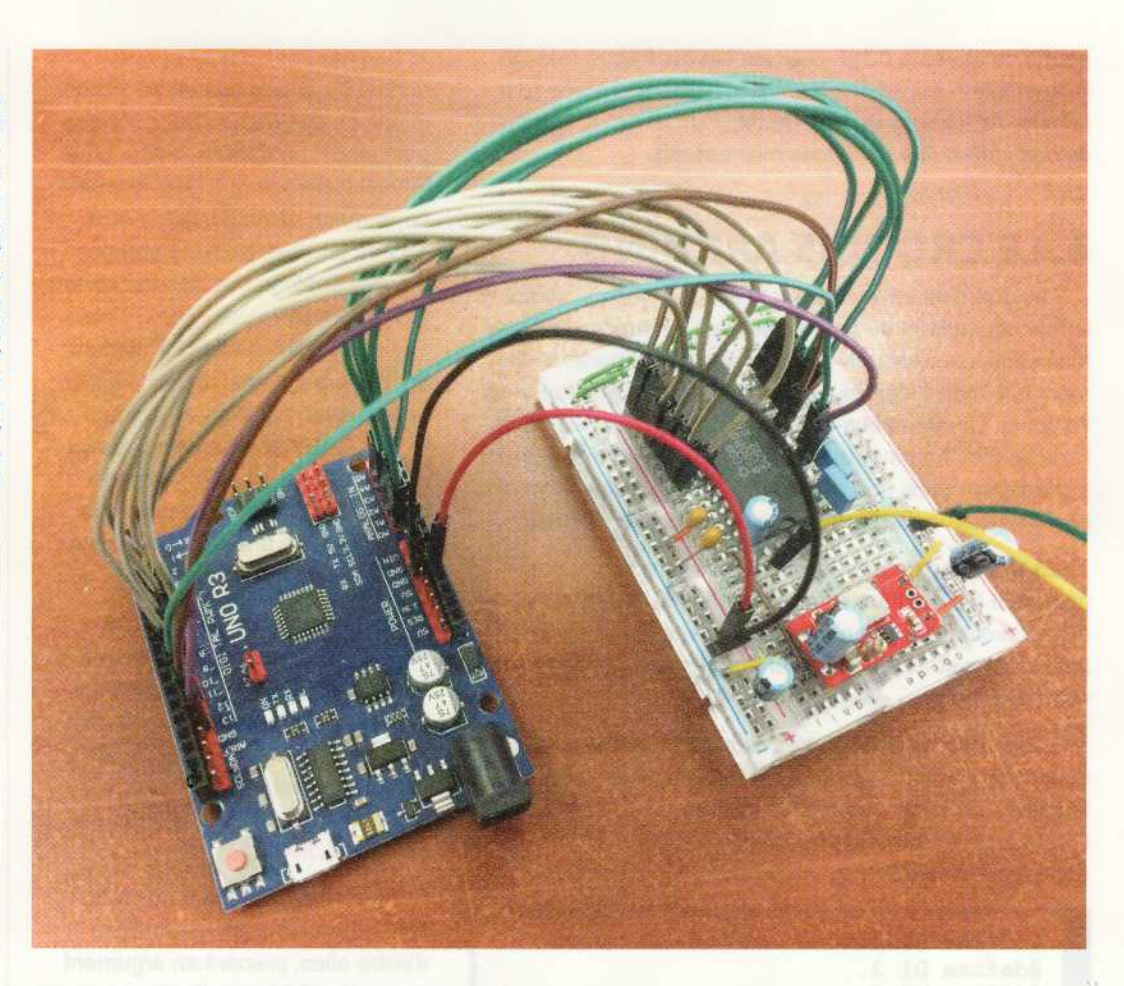
La valeur contenue dans 25 registres de 8 bits correspondant à 50 caractères en notation hexadécimale, notre tampon (*buffer*) permettant la réception des données, laconiquement appelé **buffer[]**, doit donc pouvoir accueillir 50 caractères (**char**).

Passons maintenant aux fonctions utilitaires avec la première d'entre elles, prenant en argument une valeur 8 bits ainsi qu'une adresse et écrivant la valeur dans le registre à l'adresse correspondante :

```
void poke(char addr, char val)
{
  digitalWrite(A0,(addr & 0x1));
  digitalWrite(A1,(addr & 0x2));
  digitalWrite(A2,(addr & 0x4));
  digitalWrite(A3,(addr & 0x8));
  digitalWrite(A4,(addr & 0x10));

  digitalWrite(D0,(val & 0x1));
  digitalWrite(D1,(val & 0x2));
  digitalWrite(D2,(val & 0x4));
  digitalWrite(D3,(val & 0x8));
  digitalWrite(D4,(val & 0x10));
  digitalWrite(D5,(val & 0x20));
  digitalWrite(D6,(val & 0x40));
  digitalWrite(D7,(val & 0x80));
```





Ceci est la toute première version du montage, sur platine à essais avec son convertisseur de tension extrait sauvagement de son boîtier. Ce premier essai a permis de tirer une première conclusion importante : pour l'audio, mieux vaut éviter les platines à essais.

```
    digitalWrite(CSPIN, LOW);
    delay(1);
    digitalWrite(CSPIN, HIGH);
}
```

Comme celle-ci est presque totalement équivalente à la commande **POKE** du BASIC sur C64, elle est tout naturellement appelée **poke()**. Celle-ci se résume à l'extraction de l'état de chaque bit des deux octets et aux changements d'états correspondants sur le bus d'adresses et de données, juste avant d'activer brièvement le signal /CS du SID (actif à l'état bas, comme c'est souvent le cas pour les signaux *chip select* des composants logiques). Notez qu'ici le premier registre est à l'adresse **0x0** alors que dans le cas d'un C64 c'est **0xd400**, puisque ce registre est *mappé* dans l'espace mémoire. Le C64 comporte effectivement 64 Ko de RAM, mais tout cet espace n'est pas utilisable en tant que tel, du moins pas avec la configuration par défaut (un programme peut désactiver une partie du mappage en mémoire pour divers éléments en ROM par exemple). Ce qui explique, par exemple, le message signifiant que vous avez 38911 octets disponibles au démarrage de l'ordinateur dans l'interpréteur BASIC (le langage BASIC, la table de caractères et les routines de base (KERNAL) sont en ROM, mais accessibles via l'espace mémoire de la RAM).

À présent que nous disposons d'une fonction pour envoyer des données au SID, nous pouvons en écrire une nouvelle procédant au reset et initialisant tous les registres à zéro :

```
void resetSID()
{
  char addr;
  // Reset SID
  digitalWrite(RESETPIN, LOW);
  delay(20);
  digitalWrite(RESETPIN, HIGH);
  delay(20);
  // Tous les registres inscriptibles à 0
  for(addr = 0; addr < 25; addr++)
    poke(addr, 0);
}
```

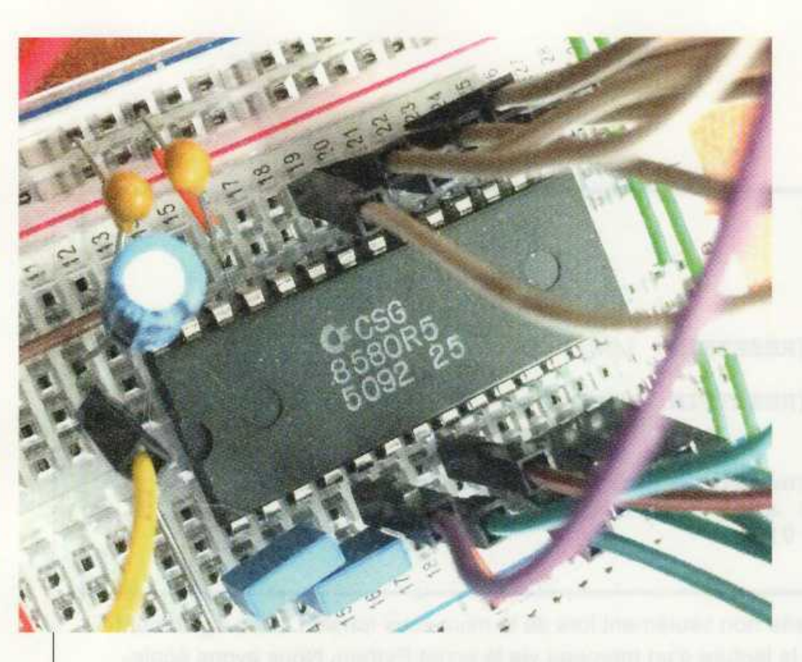
Cette fonction sera utilisée non seulement lors de la mise sous tension, mais également une fois qu'on interrompra la lecture d'un morceau via le script Python. Nous avons également besoin d'une fonction destinée à mettre à jour les registres du SID en fonction des données reçues :

```
// mise à jour de registre
void updateSID()
{
  for (int i = 0; i < 25; i++) {
    // on n'écrit que si la valeur est différente
    if (oldsid[i] != newsid[i]) {
      oldsid[i] = newsid[i];
      poke(i, newsid[i]);
    }
  }
}
```

Comme vous pouvez le voir, c'est cette fonction qui déterminera si la modification est nécessaire. Comme les 25 premiers registres, qui contrôlent la création des sons, sont en écriture seule, nous ne pouvons pas en obtenir le contenu afin de vérifier si une mise à jour est nécessaire ou non. Nous conservons donc toujours la dernière valeur inscrite dans chaque registre dans le tableau **oldsid[]** et y mettons à jour la valeur qui s'y trouve en même temps que celle placée dans le SID.

Les données reçues via le port série ne sont pas binaires, mais une suite de lettres et de chiffres. Nous ne pouvons donc pas utiliser quelque chose comme **"A4"** puisque c'est une chaîne de caractères et non la valeur **0xa4**. Pour convertir le texte en valeur, nous avons donc besoin d'une fonction supplémentaire :

```
// hex vers 4bits
char decode(char ch)
{
  if (ch >= 'A')
    return ch - 55;
  else
    return ch - 48;
}
```



Le SID 8580R5 est le modèle que vous trouverez le plus facilement et sans le moindre doute celui que vous aurez le moins de crainte à utiliser. Remarquez ici la présence de condensateurs de découplage (jaune) entre les lignes d'alimentation et la masse afin de prévenir tout bruit parasite.

Celle-ci prend en argument un caractère entre `"0"` et `"F"` et retourne un octet entre `0x00` et `0x0f`, sur la base de la table ASCII standard. Pour obtenir une valeur sur 8 bits, nous devrons donc utiliser deux fois cette fonction. Par exemple avec `"A4"`, nous l'appellerons avec `"A"` et `"4"` pour obtenir `0x0a` et `0x04` qu'il nous suffira de combiner avec un OU logique et un décalage de 4 bits vers la gauche pour la première valeur. `0x0a` devient `0xa0` qui combiné à `0x04`, devient `0xa4`.

La dernière fonction à créer, et sans le moindre doute la plus importante, est celle destinée à réceptionner les données sur le port série, les convertir et les accumuler :

```
// lecture des données
void readData()
{
  if (Serial.available() > 0) {
      char ch = Serial.read();
      // demande de reset par le script Python ?
      if (ch == 'R') {
        resetSID();
        return;
      }
      // données pour le registre ?
      if (ch == '!') {
        // données complètes ?
        if (idx == 50) {
          // décodage
          for (int i = 0; i < 25; i++) {
            char highnibble = buffer[i * 2];
            char lownibble = buffer[i * 2 + 1];
            newsid[i] = (decode(highnibble) << 4) | decode(lownibble);
          }
          // données prêtes
          dataready = true;
        }
        idx = 0;
      } else {
        // accumulation
        if (idx < 50)
          buffer[idx] = ch;
        idx++;
      }
  }
}
```

La logique utilisée par cette fonction consiste à accepter des nouveaux caractères jusqu'à obtenir **"!"** validant le « message » véhiculant la valeur des registres. Si tel est le cas, nous nous assurons que nous avons effectivement 50 caractères correspondant à 25 octets en notation hexadécimale et procédons à la conversion pour remplir **newsid[]**. Le contenu de **dataready** est alors passé à *VRAI* pour que le code dans la boucle principale (**loop()**) puisse déclencher la mise à jour des registres sur le SID.

Il ne nous reste plus maintenant qu'à écrire la fonction **setup()** pour configurer les broches et le port série, et pour procéder à la réinitialisation du SID :

```
void setup()
{
  Serial.begin(115200);
  delay(100);

  // setup reset
  pinMode(RESETPIN, OUTPUT);
  digitalWrite(RESETPIN, HIGH);
  // setup SID chip select
  pinMode(CSPIN, OUTPUT);
  digitalWrite(CSPIN, HIGH);
  // setup data lines
  pinMode(D0, OUTPUT);
  pinMode(D1, OUTPUT);
  pinMode(D2, OUTPUT);
  pinMode(D3, OUTPUT);
  pinMode(D4, OUTPUT);
  pinMode(D5, OUTPUT);
  pinMode(D6, OUTPUT);
  pinMode(D7, OUTPUT);
  // setup address lines
  pinMode(A0, OUTPUT);
  pinMode(A1, OUTPUT);
  pinMode(A2, OUTPUT);
  pinMode(A3, OUTPUT);
  pinMode(A4, OUTPUT);

  delay(20);
  resetSID();
  delay(20);
}
```

Et enfin, nous avons la fonction **loop()** qui ne fait que vérifier si des données sont prêtes, et le cas échéant, envoyer les données au SID :

```
void loop()
{
  // si données pas prêtes, continuer à lire
  if (!dataready)
    readData();
  // si données prêtes, écrire dans le SID
  if (dataready == true) {
    dataready = false;
    updateSID();
  }
}
```

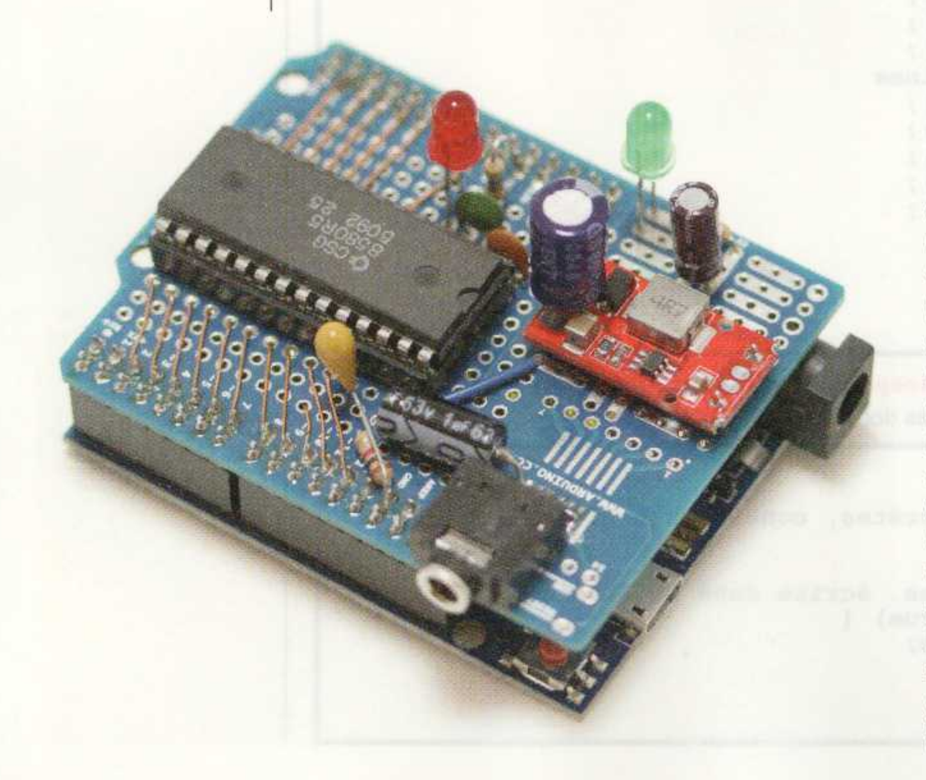
5. LE SCRIPT/LECTEUR EN PYTHON

Je serai plus bref sur le code Python que vous pourrez trouver dans son intégralité sur le dépôt GitHub du magazine, en raison de sa taille. Celui-ci repose sur l'utilisation d'un module appelé Py65, chargé de la simulation du processeur 6502. Vous pourrez installer ce module avec la commande **sudo pip install -U py65** après avoir installé le paquet **python-pip** sur votre Raspberry Pi ou votre PC Debian GNU/Linux (ou Ubuntu). Le paquet **python-serial** sera également nécessaire pour la communication avec la carte Arduino.

Si on fait l'impasse sur toute la partie traitement des arguments de la ligne de commandes, la mise en forme des messages affichés, la manipulation sur les fichiers et la gestion d'erreurs, le script se résume en quatre parties/tâches distinctes :

- analyse du fichier SID afin de déterminer ses spécificités (version, adresses de chargement et d'exécution, éléments textuels, type de cadencement, etc.) ;
- émulation du 6502 avec création d'un tableau de 65536 octets (64 Ko) constituant la mémoire du C64, et exécution des instructions en langage machine incluses dans le fichier SID avec arrêt conditionné de l'exécution ;
- inspection de la mémoire aux adresses où se trouvent mappés les 25 registres qui nous intéressent, **0xd400** à **0xd418**, mais également le cas échéant **0xdc04** et **0xdc05**, correspondant aux deux registres du CIA contenant la valeur du timer A du CIA 1 (pour le cas où le fichier SID utilise cette méthode et non la fréquence de rafraîchissement verticale de l'écran) ;
- envoi des données de façon cadencée, sous la forme de 50 caractères correspondant aux valeurs hexadécimales des 25 registres 8 bits du SID.

Le format des fichiers SID est dûment documenté et se compose de trois parties : une en-tête spécifiant les caractéristiques du fichier, un morceau de code chargé de la lecture de la musique et des données brutes propres au code. Pour pouvoir lire un fichier SID, il faut l'exécuter et nous ne savons absolument pas en quoi consiste exactement le code, ni même où il se termine et où débutent les données. C'est un peu comme si tous les fichiers MP3 contenaient un lecteur MP3 intégré avec en prime, aucune garantie qu'il s'agisse bien d'un format standard MP3.



La seconde version du montage est la déclinaison « shield de prototypage » pour Arduino, simplifiant grandement les connexions du bus d'adresse et du bus de données. Cette version utilise également une fréquence de 1 Mhz générée par la carte Arduino.

Heureusement pour nous, l'en-tête du fichier contient différentes informations parmi lesquelles le nom de l'artiste, le titre du morceau, l'adresse de chargement en mémoire, la technique utilisée pour cadencer la lecture, etc. Pour analyser un fichier, il suffit donc de lire, octet par octet, le début du fichier et donner un sens à chaque donnée pour en tirer les informations souhaitées.

Une fois toutes ces informations collectées, nous pouvons mettre en route la simulation de la machine à base de 6502. Cette simulation a besoin d'une mémoire et nous lui fournissons sous la forme d'une variable `memory` contenant `0x10000` (64 Ko) fois la valeur `0`. Un objet `cpu` peut ensuite être créé en appelant `cpu = mpu6502.MPU(memory)`. Dès lors, `cpu` représentera notre processeur simulé et nous construisons une fonction permettant de l'animer :

```
def runCPU(cpu, newpc, newa, newx, newy):
  cpu.pc = newpc
  cpu.a = newa
  cpu.x = newx
  cpu.y = newy
  cpu.sp = 0xFF

  running = True
  instructioncount = 0

  while running and instructioncount < 1000000:
    # sortie de sous-routine ?
    if cpu.ByteAt(cpu.pc) in (0x40, 0x60) and cpu.sp == 0xFF:
      running = False

    # instruction BRK ?
    if cpu.ByteAt(cpu.pc) == 0x00:
      running = False

    cpu.step()
    instructioncount += 1

    # fin de lecture ?
    if (cpu.ByteAt(0x01) & 0x07) != 0x5 and cpu.pc in (0xea31, 0xea81):
      running = False
```

L'objet de cette fonction n'est ni de procéder à une exécution pas à pas ni de laisser fonctionner le processeur seul. Notre but est d'appeler `runCPU()` jusqu'à ce que nous ayons une condition favorable à l'arrêt de la simulation, découlant du fonctionnement supposé du code de lecture inclus dans le fichier SID. Ces codes, qui sont propres aux artistes/compositeurs/programmeurs, fonctionnent généralement avec une sous-routine de lecture. L'idée est donc ici de guetter des opcodes en langage machine qui correspondent à la fin de l'exécution d'une sous-routine. Lorsque cela arrive, c'est que des données ont été envoyées au SID lors d'une lecture sur un vrai C64. C'est donc aussi le moment où arrêter la simulation et aller voir en mémoire ce qui est inscrit dans les registres du SID.

Nous n'avons pas besoin d'émuler le SID puisque ces registres sont *mappés* en mémoire. Le code du fichier SID a simplement écrit à certaines adresses et, contrairement à une situation réelle avec du matériel, nous pouvons aller lire ces valeurs. Comme la mémoire n'est rien d'autre d'un grand tableau de 64 Ko, pour obtenir la valeur des registres, il nous suffit de lire 25 octets de la position `0xd400` à `0xd418`. Si l'en-tête du fichier SID indique que le CIA1 est

utilisé pour cadencer la lecture, nous faisons de même avec les valeurs aux adresses **0xdc04** et **0xdc05**. Ici se trouve la valeur du Timer 1 sur 16 bits correspondant au temps devant s'écouler entre deux appels à la sous-routine de lecture. Comme le C64 fonctionne à 1 Mhz, chaque incrément de 1 de cette valeur correspond à un délai de 1 µs. Si le CIA1 n'est pas utilisé, c'est une fréquence de 50 Hz qui l'est, soit un délai de 1s/50 = 20 ms (ou 20000 µs).

Ce qui nous amène à la dernière partie du code. Comme nous avons choisi de confier au code Python la tâche de cadencer la lecture, nous nous heurtons à un petit problème : Python n'est absolument pas adapté à ce genre de choses. Pour tout dire, ceci serait plus facile avec un autre langage, comme le C, mais loin d'être parfait pour autant. Il y a une énorme différence entre un code s'exécutant sur une carte Arduino et sur un système plus complexe comme un PC : l'Arduino ne fait fonctionner qu'un seul code à la fois et il est possible d'obtenir très précisément des délais. Avec un PC faisant fonctionner un système multitâche, le fonctionnement d'un programme peut être ralenti ou accéléré en fonction de la charge de la machine. Un système capable de gérer correctement ce genre de choses est appelé un système d'exploitation temps-réel, chose qu'un système standard pour PC n'est pas.

Pour arriver à nos fins, il faut ruser et la solution finalement adoptée consiste simplement à chronométrer toute la partie de récolte des valeurs des registres (avec l'appel à **runCPU()**), jusqu'à l'envoi des données sur le port série. Ceci peut se faire avec **datetime.datetime.now()** qui peut être vu comme le **millis()** d'Arduino appliqué à Python. Nous relevons la date/heure avant l'opération, puis après, et nous en déduisons le temps écoulé. Comme celui-ci est normalement inférieur à la période entre deux envois de données, nous complétons la différence avec **time.sleep()**. La précision n'est pas parfaite, mais bien suffisante à l'échelle où nous travaillons. Une cadence de 50,02 Hz, par exemple, en lieu et place de 50,00 Hz ne sera pas réellement audible lors de la lecture.

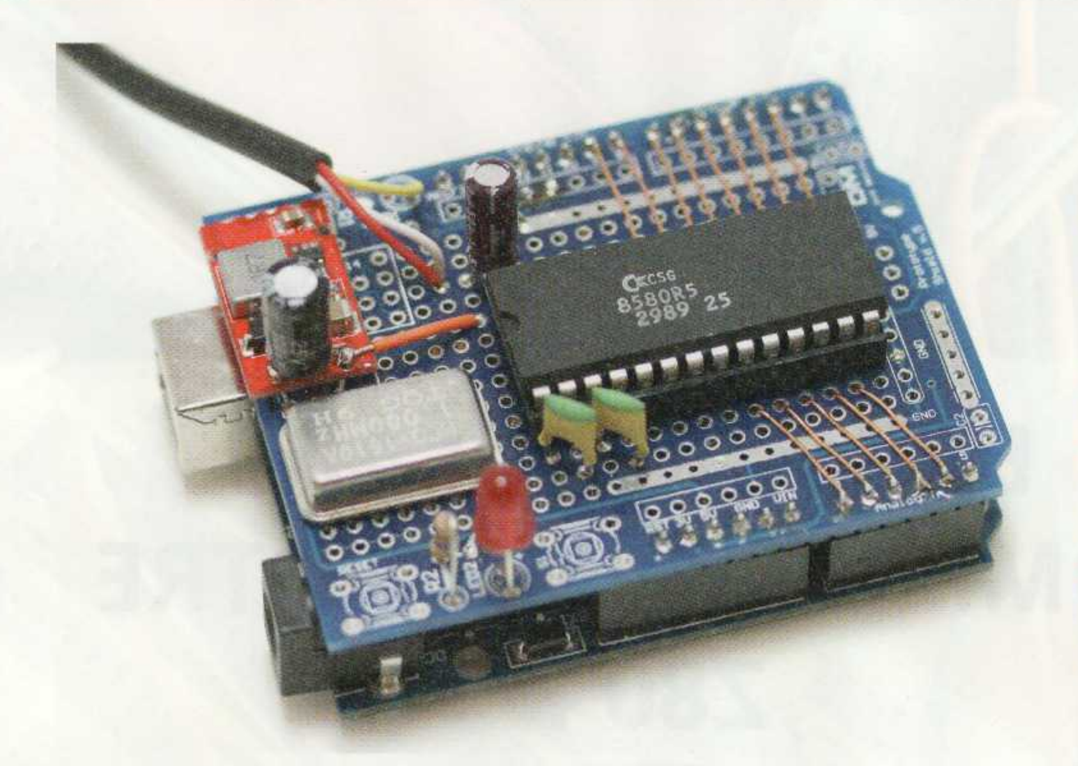
L'alternative à une telle approche aurait pu être de conserver la gestion de la temporisation côté Arduino tout en intégrant un nouveau type de message envoyé par le script Python de façon à faire varier le délai (fonction de la valeur du timer du CIA). Personnellement, je préfère que le montage Arduino s'en tienne à être un simple périphérique, comme une carte son USB, ne contrôlant donc pas de processus de lecture. L'une et l'autre approche ayant leurs avantages et leurs inconvénients, force est de constater que le problème est en réalité une conséquence directe de la division du projet en une partie PC et une partie Arduino.

Comme spécifié précédemment, vous trouverez le code Python ainsi que le croquis Arduino de ce projet sur le dépôt GitHub de ce numéro. Il vous suffira alors de charger le croquis dans la mémoire de la carte UNO puis d'utiliser le code Python en spécifiant le port série à utiliser (**-p**), éventuellement un mode verbeux (**-v**) et, bien entendu le chemin vers le fichier SID à lire. Notez que si vous ne disposez pas, ou pas encore, de SID pour votre montage, le lecteur en Python comme le croquis fonctionneront tout aussi bien.

POUR FINIR

Depuis la fin de la rédaction de cet article, mes expérimentations n'ont pas cessé, mais bien que l'idée d'un lecteur de fichiers SID faisant usage d'un montage externe en USB en guise de « carte son » soit très séduisante, une autre, bien plus plaisante à mes yeux, s'est fait jour. Celle-ci a découlé d'une simple question : comme **siddump** émule très bien la partie du C64 qui m'intéresse, et ce en quelques 1200 lignes de C (50% macros, 50% un gros **switch**/**case**), ne serait-il pas possible de fait fonctionner l'ensemble du projet sur une carte... et sans PC ?

Bien entendu, une carte Arduino comme la UNO n'est pas capable de faire une telle



Voici la version finale du projet avec l'oscillateur à quartz et un meilleur agencement des composants (certaines connexions se font par le dessous du shield). Il n'y aura sans doute plus d'autres déclinaisons du montage à l'avenir puisque mon objectif est à présent de rendre tout cela totalement autonome (sans PC, Mac ou Pi) en reposant sur un ESP32 et une carte micro SD pour le stockage de fichiers...

chose faute de mémoire et de ressources CPU, mais avec quelque chose d'autre... basé sur un ESP32... c'est une tout autre histoire ! Ces cartes, comme nous l'avons vu dans un précédent numéro, sont riches en fonctionnalités, mais surtout, disposent de la « puissance » nécessaire pour simuler un processeur 6502/6510 à une cadence plus que raisonnable, de la mémoire pour charger le contenu des fichiers, et des périphériques complémentaires permettant à la fois de dialoguer avec le SID, mais aussi de s'ouvrir à d'autres possibilités : contrôle en Wifi, stockage sur SD, interfaçage avec un écran SPI, animation lumineuse en fonction de l'utilisation des voix et des registres, etc.

À l'heure où le présent texte est couché sur papier, le code permettant de charger, lire et « exécuter » un fichier SID fonctionne d'ores et déjà sur ESP32. L'interface avec le SID ne devrait pas poser de problème en dehors de l'ajustement des niveaux de tensions (l'ESP32 est 0/+3,3V uniquement et le SID 0/+5V). Dans le futur, vous entendrez probablement parler de ce nouveau projet, d'une façon ou d'une autre, mais à ce stade, les choses se présentent plutôt bien même s'il reste énormément de travail.

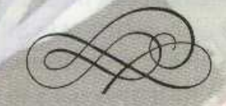
Mon projet de lecteur de chiptunes va donc se transformer en projet de baladeur à chiptunes, mais d'autres déclinaisons peuvent être envisagées. Le SID n'est pas le seul composant générateur de sons utilisable (même si c'est le plus remarquable à mon avis), et tout ce qui vient d'être détaillé, même si relativement spécifique aux technologies MOS/Commodore, pourra être transposé vers le TIA d'un Atari 2600, le General Instrument AY-3-8910, ainsi que toute une collection de puces Yamaha (dont le YM3812 de la toute première Sound Blaster).

Une autre approche peut également consister en l'oubli pur et simple de la lecture de fichiers, afin de construire un instrument de musique de toutes pièces, éventuellement assorti d'une interface MIDI. Là encore, il y a beaucoup à faire, mais c'est un domaine qui demande quelque chose qui, personnellement, me fait cruellement défaut : une oreille musicale. Il n'existe malheureusement pas de montage permettant de palier à ce type de carence, mais peut-être n'êtes-vous pas dans ce cas et ferez de votre (ou vos) SID un synthétiseur tout à fait remarquable... DB



ET SI ON FAISAIT COMMUNIQUER NOTRE Z80 ?

Denis Bodor



Dans les articles précédents, nous avons découvert le Z80, programmé en langage machine puis en assembleur et enfin nous avons écrit notre premier code C pour le voir être compilé par SDCC et exécuté par notre cher processeur Z80. Avec ces progrès non négligeables, nous sommes à présent bien équipés pour réellement faire quelque chose avec le Z80. Nous allons donc lui adjoindre un composant pour lui offrir un port série et pouvoir enfin le laisser s'exprimer joyeusement. Au boulot !

Le code en C que nous avons pu utiliser la dernière fois était relativement simple, voire totalement inutile si ce n'est pour nous assurer que tout fonctionnait et que le compilateur avait correctement fait son travail. Mais il faut bien l'avouer, observer l'exécution du code sautant d'adresse mémoire en adresse mémoire n'est pas ce qu'il y a de plus démonstratif. De plus, je le rappelle, nous avions soigneusement évité d'utiliser des variables globales initialisées puisque nous n'avions rien prévu dans ce sens. Au menu de cet article relativement dense donc, nous commencerons par corriger ce défaut et nous nous attaquerons à l'ajout d'un port série au montage ou plus exactement un UART, pour *Universal Asynchronous Receiver/Transmitter*.

1. RÉGLONS LE PROBLÈME DES VARIABLES GLOBALES

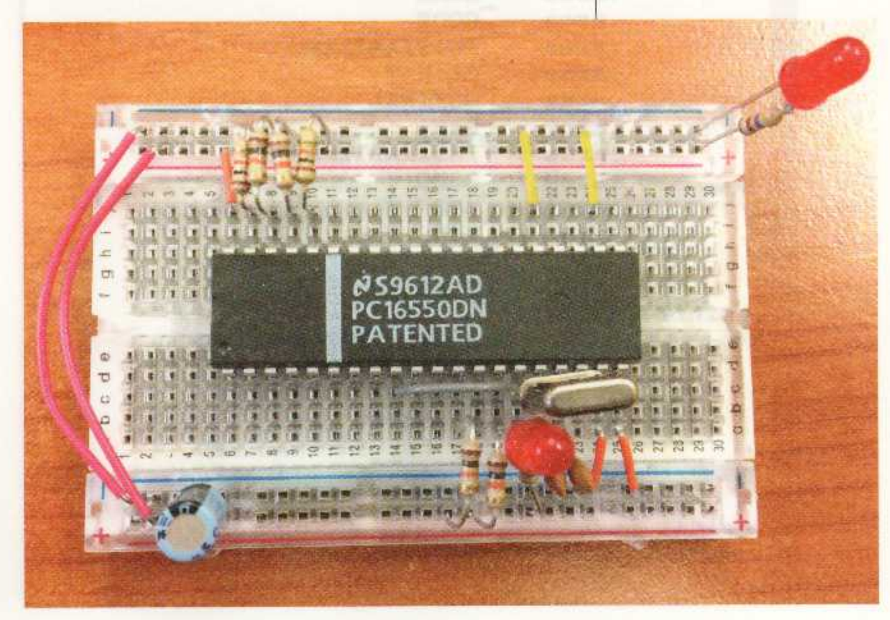
Du point de vue d'un processeur, une variable n'est rien d'autre que quelque chose à une certaine adresse mémoire, un emplacement où se trouve une donnée ou le début d'une donnée. Dans l'article précédent, nous avons pu voir qu'une variable globale comme `uint8_t toto` se résume à huit malheureux bits placés quelque part en mémoire. Nous nous sommes amusés à déclarer cette variable, l'avons initialisée avec la valeur `0x42` au début de la fonction `main()` et avons appelé une fonction `essai()` prenant en argument une valeur devant s'additionner à celle présente dans `toto`. Ce que nous n'avons pas pu faire, en revanche, c'est :

```
uint8_t toto = 0x42;
```

Ceci n'a l'air de rien, mais pour le compilateur c'est un vrai problème. En effet, le point d'entrée de notre programme en C est la fonction `main()`, mais si celle-ci tente de lire `toto`, on s'attendra normalement à obtenir `0x42`. Il faut donc que quelque chose ait placé cette valeur à l'adresse désignée par `toto` et ce quelque chose est, bien entendu, le crt0 avec lequel nous avons déjà fait connaissance. Seulement voilà, notre déclaration de `toto` et son initialisation se trouvent dans le fichier C et le crt0 n'en a pas connaissance.

La solution peut sembler bien complexe, mais il n'y en a pas d'autre et consiste à prévoir dans le crt0 une routine qui va lire une adresse mémoire pour obtenir l'emplacement de la variable à initialiser (en RAM donc) et une autre pour la valeur qui doit prendre place dans la variable. Bien entendu, cette valeur d'initialisation doit déjà exister lors de l'exécution du code. Nous avons donc une variable en RAM et sa valeur en ROM, qui doit être copiée avant l'appel à `main()`.

Voici le composant que nous interfaçons ici avec notre Z80. Il s'agit d'un UART 16550 de National Semiconductor que l'on peut sommairement qualifier de « port série ». Normalement, avec un Z80, on utiliserait un Zilog Z84C4008 spécialement dédié à une telle tâche par le fabricant du processeur et non une puce initialement dédiée aux PC... mais pourquoi se priver d'un peu d'originalité ?







Le 16550 ou même le Z84C4008 ne sont de loin pas les seuls composants pouvant offrir des fonctionnalités de communication série à notre projet. Voici, par exemple, des KR580VV51A, des clones soviétiques de l'Intel 8251, parfaitement compatibles avec un Z80 de Zilog et parfaits pour créer un ordinateur 8 bits avec un petit aspect « uchronique » mélangeant des technologies ne s'étant jamais rencontrées dans l'Histoire...

Heureusement, pour placer et retrouver tout ce petit monde, nous pouvons compter sur l'éditeur de liens, à condition d'utiliser, dans notre crt0, des directives précisant des emplacements relatifs qu'il se chargera, ensuite, de faire correspondre à nos besoins. Dans le précédent article, nous avions déjà utilisé une telle directive pour indiquer l'emplacement absolu du début du code (`_HEADER (ABS)`). Il nous suffit alors de compléter cela en précisant les autres segments/sections pour que l'éditeur de liens puisse faire son travail, puis d'ajouter notre routine d'initialisation. Voici donc notre nouveau `crt0.s` :

```
        .module crt0
        .globl   _main

        .area    _HEADER (ABS)
        .org     0x0
init:
        ld      sp,#0x0200

        call    gsinit
        call    _main

        ;; segments pour l'éditeur de liens
        .area   _HOME
        .area   _CODE
        .area   _INITIALIZER
        .area   _GSINIT
        .area   _GSFINAL
        .area   _DATA
        .area   _INITIALIZED
        .area   _BSEG
        .area   _BSS
        .area   _HEAP

        .area   _GSINIT
gsinit::
        ld      bc, #l__INITIALIZER
        ld      a, b
        or      a, c
        jr      Z, gsinit_next
        ld      de, #s__INITIALIZED
        ld      hl, #s__INITIALIZER
        ldir
gsinit_next:
        .area   _GSFINAL
        ret
```

Tout ce qui se trouve après `call _main` est nouveau, tout comme l'appel à `call gsinit` pour exécuter la routine d'initialisation avant `main()`. Notez également que le registre SP est ici initialisé avec la valeur `0x200` (512), c'est-à-dire l'adresse juste après la fin de la mémoire que nous allons émuler. Vous l'avez compris, cette fois nos 64 malheureux octets de mémoire ne suffiront pas, pour notre présente évolution, nous avons besoin de 512 octets.

C'est notre routine `gsinit:` qui sera chargée d'initialiser les variables globales grâces aux symboles `l__INITIALIZER`, `s__INITIALIZED` et `s__INITIALIZER`. *INITIALIZER* désigne la zone contenant les valeurs devant être utilisées (en ROM) et *INITIALIZED* la zone en RAM correspondant aux variables devant être initialisées. Les symboles débutant par `l_` désignent une taille (« l » pour *length*) et ceux débutant par un `s_` désignent un emplacement en mémoire (« s » comme *start*). Ces symboles, comme d'autres, pouvant être observés dans le fichier `.map` résultant de la compilation, sont gérés et utilisés par SDCC pour réallouer dynamiquement nos éléments, lors de l'édition de liens.

Notre routine d'initialisation consiste en une simple copie d'un

morceau de mémoire, d'une taille `l__INITIALIZER`, vers un autre, de `s__INITIALIZER` à `s__INITIALIZED`, via l'instruction `ldir` (copie répétitive des données débutants à par HL, vers DE, le nombre de fois indiqué par BC).

Dernier élément important, vous remarquerez que l'étiquette `gsinit` se termine par deux double-point et non un seul, ceci en fait un symbole global qui sera donc connu en dehors de ce simple fichier/module. Ceci est indispensable pour que l'éditeur de liens puisse appeler notre routine le moment voulu.

Vous pouvez déjà utiliser ce nouveau `crt0.s` avec le code du précédent article, mais ceci nécessitera quelques changements dans le fichier `Makefile` car, pour que le compilateur et l'éditeur de liens puissent faire leur travail, il faut spécifier les fichiers dans le bon ordre pour la construction. De plus, cette routine `gsinit:` occupe de la place (`0xf` octets) et fait donc grossir notre code qui ne tient déjà plus dans 64 octets de mémoire émulée si l'on souhaite laisser de la place pour la pile.

2. AJOUT PHYSIQUE DE L'UART

Ajouter un périphérique géré par le processeur Z80 n'est, en théorie, pas très compliqué. En effet, de la même manière que nous avons les lignes /MREQ, /RD et /WR permettant au processeur de signifier un accès à la mémoire, en lecture ou en écriture, nous avons /IORQ (broche 20) qui, lorsqu'il est à la masse, indique un accès vers une adresse en entrée/sortie, ou E/S (I/O en anglais pour *Input/Output*). Lorsque cela arrive, la moitié basse du bus d'adresse présente une adresse E/S et non une adresse mémoire.

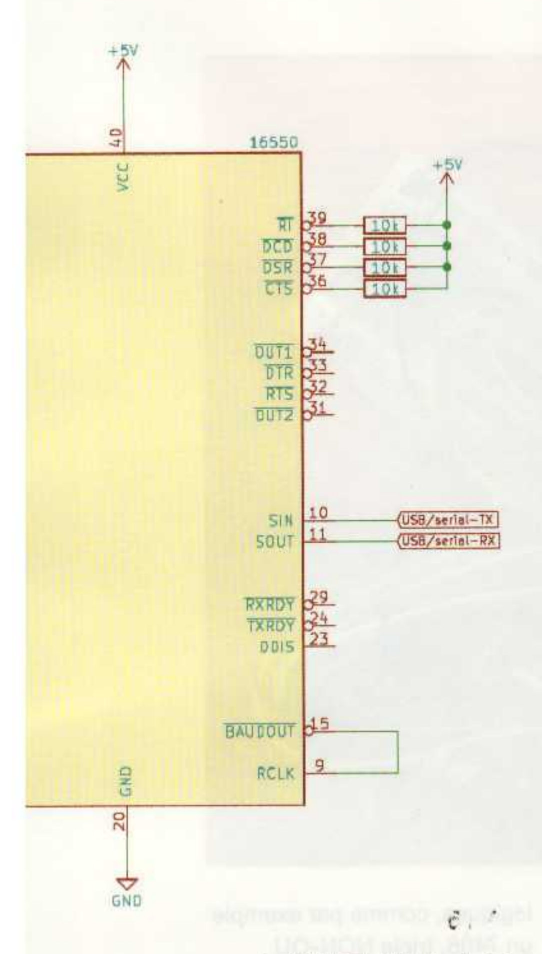
Du point de vue logiciel, il est relativement simple d'utiliser cette fonctionnalité puisque ceci se résume à l'utilisation de l'instruction `OUT (port),a` où `port` est une valeur immédiate correspondant à une adresse qui sera présentée sur le bus d'adresse et où `a` contient la donnée qui sera présentée sur le bus de données. L'instruction opposée, `IN`, a pour opérandes un registre et le port sous la forme, par exemple de `IN a,(port)`.

Ce mécanisme permet de séparer l'accès à la mémoire de l'accès à un autre composant. On ne parle plus alors d'adresse, mais de port, ou tantôt de port E/S. La moitié du bus d'adresse de 16 bits correspond à un espace adressable de 256 adresses (2 puissance 8), mais ceci ne signifie pas qu'il est possible d'accéder à 256 composants de cette manière. Celui qui nous occupe ici est un UART 16650 (successeur du 8250 avec lequel il est compatible), il possède un bus de données de 8 bits et trois lignes d'adresses A0 à A2, permettant d'accéder à 8 registres internes utilisés pour

la configuration et pour la communication série. Nous consommons donc 8 des 256 adresses avec un seul composant.

Physiquement, le 16550 (comme le 8250) se présente sous la forme d'un composant à 40 broches :

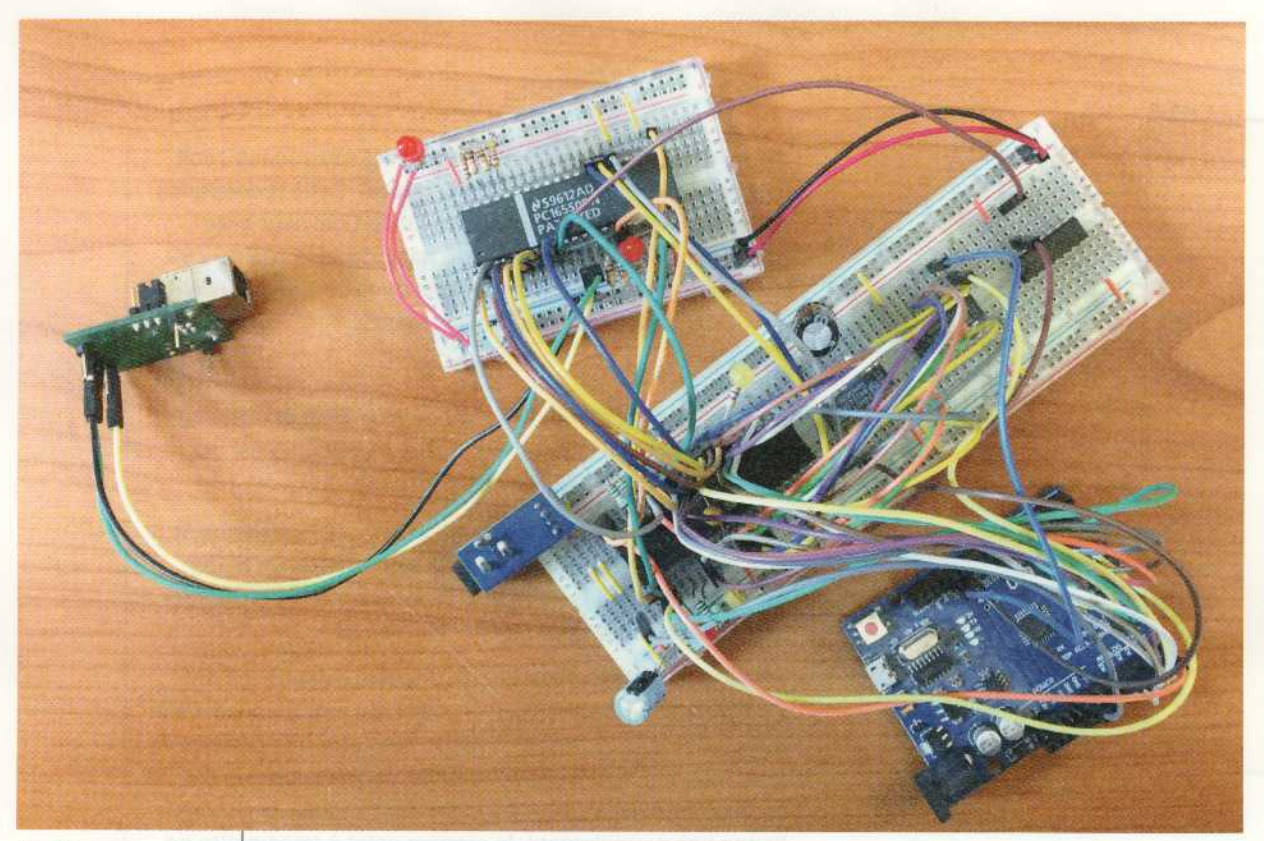
- D0 à D7 (1 à 8) : bus de données relié aux lignes correspondantes du Z80.
- RCLK (9) connecté à /BAUDOUT (15) : la première broche est une entrée pour un signal d'horloge permettant de cadencer la réception de données série (16x la fréquence du débit en bauds) et la seconde le signal similaire, généré en interne, pour l'émission. En reliant ces deux broches, la vitesse d'émission et de réception sera identique.
- SIN (10) et SOUT (11) sont respectivement les lignes de réception et d'émission des données série avec des niveaux logiques 0/+5V. C'est ici que nous connecterons un convertisseur USB/série pour voir les messages s'afficher.
- CS0 à /CS2 (12 à 14) est le *Chip Select* du composant permettant de s'adresser à lui. Le 16550 est sélectionné lorsque CS0 et CS1 sont à l'état haut et /CS2 à l'état bas. Ici nous connecterons CS0 et CS1 à la tension d'alimentation via deux résistances de 10 KOhms et /CS2 sera connecté à la broche /IORQ du Z80.
- XIN (16) et XOUT (17) permettent la connexion d'un quartz pour générer la fréquence de communication



de l'UART. J'ai ici choisi un quartz à 14,7456 Mhz associé à deux condensateurs céramiques de 18 pF connectés entre les broches et la masse. Le circuit oscillant formé avec le 16550 doit être d'une fréquence correspondant à une puissance de deux. Un registre particulier du 16550 est utilisé pour diviser la fréquence afin d'obtenir le nombre de bauds auquel l'UART communique, selon la formule : *diviseur = hertz/(baud x 16)*. Ici 14745600/(9600×16) = 96, pour une communication à 9600 bauds. Cette valeur devra être précisée dans un registre du 16550 avant son utilisation.

- /WR (18) et WR (19) sont les deux broches permettant d'indiquer une écriture lorsque le 16550 est sélectionné : soit WR est au +5V, soit /WR est à la masse. Cette redondance permet au 16550 d'être piloté avec un processeur avec des signaux actifs à l'état bas ou haut. Dans le cas du Z80, nous avons /WR, nous connectons donc ce signal au /WR du 16550 et plaçons son WR à la masse.
- VSS (20) : la masse.
- /RD (32) et RD (22) fonctionnent comme /WR et WR, mais pour lire le 16550. Là encore c'est le /RD du Z80 qui est relié au /RD du 16550 et RD à la masse.
- DDIS (23) est un signal indiquant lorsque le processeur lit une donnée de l'UART. Ceci peut être utilisé si un buffer est en place entre l'UART et le processeur afin de changer la direction du flux de données. Inutilisé ici.
- /TXRDY (24) et /RXRDY (29) permettent respectivement au 16550 de signaler lorsque l'UART est prêt à transmettre ou recevoir des données. /TXRDY peut être très pratique pour mettre au point notre montage, en particulier en cas de problème avec l'oscillateur. En effet, si celui-ci ne fonctionne pas correctement, le premier octet à transmettre ne quitte pas le composant et ce signal se retrouve toujours à la masse, indiquant que l'UART n'est pas prêt à accepter d'autres données à transmettre. Le 8250, prédécesseur du 16550n ne dispose pas de ces signaux (respectivement CSOUT et non-connecté), c'est la seule différence de brochage entre les deux UART.
- /ADS (25) pour *ADdress Strobe* permet de valider la présentation d'une adresse sur les broches A0 à A2, et l'état de CS0 à /CS2, si le processeur n'arrive pas à stabiliser correctement ces signaux durant une opération de lecture ou d'écriture. Ici, nous n'en avons pas besoin et cette broche est reliée à la masse.
- A0 (28) à A2 (26) est le bus d'adresse du 16550 permettant d'accéder aux registres. Dans notre configuration, ces trois broches sont reliées aux lignes correspondantes du Z80.
- INTR (30) est la ligne d'interruption permettant au 16550 de réclamer l'attention du processeur. Nous n'utilisons pas ici d'interruption, cette broche n'est pas connectée.
- /OUT2 (31) et /OUT1 (34) sont deux sorties contrôlables de façon logicielle à la discrétion de l'utilisateur pour divers usages. Nous ne nous en servirons pas ici, mais ceci peut être utile en cas de problème avec le code.





La mise en œuvre du montage est quelque peu acrobatique, mais, avec de la patience et une bonne concentration, on arrive au résultat escompté. Nous avons ici de gauche à droite, le convertisseur USB/série, l'UART, la platine du processeur et la carte Arduino pilotant le tout.

- /RTS (32) et /DTR (33) correspondent aux signaux de contrôle de flux *Request to Send* et *Data Terminal Ready* généralement utilisés par les modems. Ces sorties ne sont pas utilisées ici et sont donc laissées déconnectées.
- /CTS, /DSR, /DCD et /RI (36 à 39) sont également des signaux, en entrée cette fois, en rapport avec les gestions des modems avec, dans l'ordre, *Clear to Send, Data Set Ready, Data Carrier Detect* et *Ring Indicator.* Comme il s'agit d'entrée active à l'état bas, nous connectons ces quatre signaux inutilisés à la tension d'alimentation avec 4 résistances de rappel de 10 KOhms.
- MR (35) est la ligne de réinitialisation (*Master Reset*) du 16550. Ce signal étant actif à l'état haut, ceci nous pose un problème. Le signal de /RESET du Z80 est actif à l'état bas et nous ne pouvons donc pas connecter ensemble ces deux signaux, à la broche A0 de la carte Arduino. Deux options s'offrent à nous, piloter le reset du 16550 avec une autre broche de l'Arduino, mais nous en manquons déjà, ou utiliser un circuit logique pour inverser le signal /RESET. J'ai ici utilisé un sextuple inverseur 74LS14 de récupération, mais d'autres composants logiques, comme par exemple un 7406, triple NON-OU comme le 7427 ou même un montage à base de transistor peut faire l'affaire. L'important est de faire d'un 0 un 1, et inversement.
- VDD (40) : tension d'alimentation +5V.

Vous l'avez sans doute compris, l'adjonction d'un UART comme ce 16550 ne nécessite pas beaucoup de composants, mais génèrera une véritable salade de câbles. Il est relativement facile de rater une connexion ou d'oublier une mise à la masse, ce qui rend très difficile et fastidieuse une recherche en cas de problème. Un autre point sensible auquel je me suis heurté est la stabilité de l'oscillateur à quartz et ce n'est qu'après une longue période, et l'ajout

d'une led sur /TXRDY que je me suis rendu compte que le composant ne pouvait pas envoyer de données. Aucun signal n'était présent sur /BAUDOUT et un remplacement des condensateurs céramiques de 33 pF en 18 pF a réglé le problème. Dans tous les cas, un conseil, ne désespérez pas si cela ne fonctionne pas du premier coup, vérifiez et revérifiez chaque connexion et, en fonction de votre équipement, assurez-vous que les tensions soient stables et tous les signaux soient corrects (un oscilloscope n'est pas indispensable, mais très utile).

3. MODIFICATION DU CODE POUR SUPPORTER LE 16550

Pour utiliser l'UART dans notre code, nous avons besoin de trois choses : configurer le 16550 après sa réinitialisation, créer une routine assembleur permettant son utilisation et mettre en œuvre des fonctions mises à notre disposition par les bibliothèques livrées avec SDCC.

Ce dernier point est important, car nous ne comptons pas réécrire les fonctions standards du C si celles-ci existent déjà. Commençons par la fin avec le code C que nous allons utiliser :

```
#include <stdint.h>
#include <stdio.h>

uint8_t toto = 0x42;

void main(void)
{
  while(1) {
    putchar ((toto-(toto%100))/100)+48 );
    putchar( (((toto%100)-(toto%10))/10)+48 );
    putchar( (toto%10)+48 );
    putchar(' ');
    puts("Coucou Hackable !\r");
    toto++;
  };
}
```

Afin d'économiser de l'espace mémoire, nous ne pouvons pas utiliser le classique **printf()** du C, ceci découlerait sur un code compilé de quelques 3 Ko. Nous devons donc trouver une autre solution pour afficher une valeur numérique correspondant au contenu de **toto** suivie d'une chaîne de caractères. Nous utilisons alors des opérations arithmétiques simples (addition, division entière et modulo) pour déduire une représentation ASCII de la valeur et complétons avec le texte statique. Les deux fonctions utilisées sont **putchar()** pour écrire un caractère et **puts()** pour faire de même avec une chaîne.

puts() provient des bibliothèques livrées avec SDCC, tout comme les routines concernant les opérations mathématiques. Mais il ne s'agit que de fonctions « génériques » n'ayant aucun lien direct avec notre sortie série. **puts()**, comme d'autres fonctions fournies, utilise **putchar()** que nous devons créer, et donc implémenter.

Et c'est précisément là que nous arrivons à l'écriture du fichier le plus important de cette étape du projet, **putchar.s** :

```
; macros
UART0  .EQU    0x00    ; DATA IN/OUT
UART1  .EQU    0x01    ; CHECK RX
UART2  .EQU    0x02    ; INTERRUPTS
UART3  .EQU    0x03    ; LINE CONTROL
UART4  .EQU    0x04    ; MODEM CONTROL
UART5  .EQU    0x05    ; LINE STATUS
UART6  .EQU    0x06    ; MODEM STATUS
UART7  .EQU    0x07    ; SCRATCH REG.

        .module putchar
        .area   _CODE

; Prend un caractère sur la pile
; et l'envoi via l'UART
_putchar::
        ld      hl,#2
        add     hl,sp
sendchar_s:
        in      a,(UART5)
        bit     #5,a
        jp      z,sendchar_s
        ld      a,(hl)
        out     (UART0),a
        ret
```

Parmi les huit registres du 16550 seul le premier sert à envoyer et recevoir des données, octet par octet, les autres permettent de configurer le composant. Cependant, nous ne pouvons pas écrire dans ce registre n'importe comment et devons nous assurer que l'UART est en mesure d'accepter une donnée de notre part avant de faire quoi que ce soit. Pour cela, nous devons consulter l'état du bit 5 du sixième registre, appelé *Transmitter Holding Register Empty (THRE) indicator* et utilisons donc l'instruction **IN** pour copier dans le registre A du Z80 la valeur présente dans le registre 5 du 16550. L'instruction **BIT** est ensuite utilisée pour tester le bit 5 et tant que celui-ci n'est pas 0, nous bouclons, en attente. S'il est à 1, c'est que le 16550 est disponible et nous envoyons la donnée se trouvant à l'adresse précédemment récupérée sur la pile, au premier registre du 16550. L'octet est transmis par l'UART.

En résumé, nous avons maintenant notre **putchar()**, notre gestion des variables globales initialisées et notre code en C. Il ne nous manque plus qu'un seul élément : configurer notre 16550. Ceci peut intervenir n'importe quand, à partir du moment où c'est avant l'utilisation effective de l'UART. Nous pourrions le faire en C, mais il est bien plus logique et économique (en termes de mémoire) d'inclure cela, via quelques lignes d'assembleurs, directement dans le crt0. Nous incluons donc les macros déjà utilisées précédemment pour désigner les registres du 16550 et ajoutons ces quelques lignes entre l'initialisation du pointeur de pile (SP) et l'appel à **gsinit** :

```
ld      a,#0x80     ; Accès au diviseur (DLAB)
out     (UART3),a
ld      a,#96       ; Diviseur= 96 = 9600bps @ 14,7456 Mhz
out     (UART0),a
ld      a,#0x00
out     (UART1),a
ld      a,#0x03     ; 8 bits de données, 1 bit de stop, reset DLAB
out     (UART3),a
```

La configuration du 16550 concerne principalement la définition de la valeur du diviseur permettant d'obtenir un débit donné à partir de la fréquence du quartz utilisé. Pour spécifier cette valeur, il faut tout d'abord passer l'UART dans un mode particulier en activant le bit DLAB (*Divisor Latch Access Bit*) du registre 3, puis en utilisant les registres 0 et 1 pour préciser une valeur sur 16 bits correspondant à celle du diviseur choisi. Ici, cette valeur est 96, nous avons donc 96 et 0 pour former les octets `0x60` et `0x00` (`0x0060` sur 16 bits et donc 96 en décimal). On configure ensuite le nombre de bits de données et de stop (pas de parité), tout en désactivant le bit DLAB, et les registres peuvent alors à nouveau être utilisés normalement.

Nous avons maintenant quatre fichiers pour notre projet, `crt0.s`, `putchar.s`, `prem.c` et enfin le `Makefile` :

```
MEMSIZE := 0x200
TARGET  := prem
CC      := sdcc
AS      := sdasz80
LFLAGS  := -Wl-u
MFLAGS  := -mz80 --code-loc 0x001C \
           --data-loc 0x0000 --vc --verbose

all: ${TARGET}.bin

crt0.rel: crt0.s
	$(AS) -plosgffw crt0.rel crt0.s

putchar.rel: putchar.s
	$(AS) -plosgffw putchar.rel putchar.s

${TARGET}.rel: ${TARGET}.c
	$(CC) -mz80 -c ${TARGET}.c

${TARGET}.ihx: ${TARGET}.rel crt0.rel putchar.rel
	$(CC) $(MFLAGS) $(LFLAGS) --no-std-crt0 crt0.rel \
	${TARGET}.rel putchar.rel -o ${TARGET}.ihx

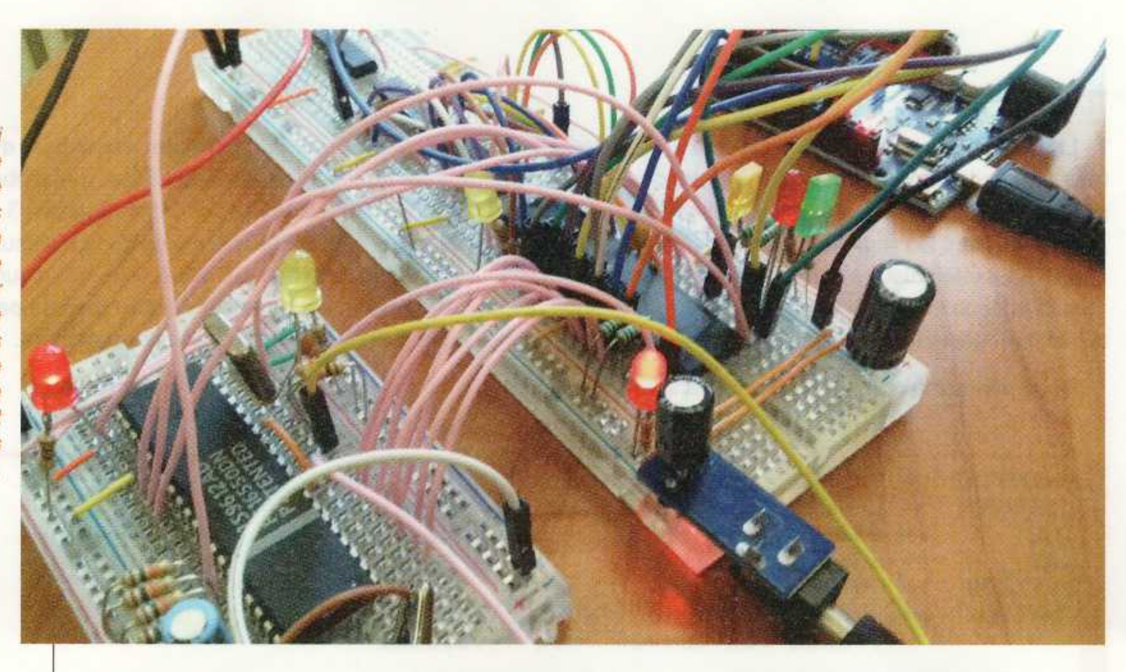
${TARGET}.bin: ${TARGET}.ihx
	makebin -p < ${TARGET}.ihx > ${TARGET}.bin
	objcopy -Iihex -Obinary --gap-fill 0x00 --pad-to \
	$(MEMSIZE) ${TARGET}.ihx ${TARGET}_padded.bin

disasm: ${TARGET}.bin
	z80dasm -t -g 0x0 -l ${TARGET}.bin

dump: ${TARGET}.bin
	xxd -c 16 -i ${TARGET}.bin

paddump: ${TARGET}.bin
	xxd -c 16 -i ${TARGET}_padded.bin

clean:
	rm -f ${TARGET}.asm ${TARGET}.cdb ${TARGET}.rel \
	${TARGET}.hex ${TARGET}.ihx ${TARGET}.lst ${TARGET}.map \
	${TARGET}.o ${TARGET}.rst ${TARGET}.sym ${TARGET}.lnk \
	${TARGET}.lib ${TARGET}.bin ${TARGET}.mem ${TARGET}.lk \
	${TARGET}.noi
	rm -f *.lst *.rel *.rst *.sym *.bin
```



On touche ici aux limites de ce qu'il est possible de faire avec des platines à essais. Il va être temps d'envisager une autre approche nous permettant de mettre en œuvre plus facilement d'autres éléments et périphériques.

Celui-ci a été révisé pour non seulement prendre en compte la mémoire augmentée à 512 octets et pour assembler `'putchar.s`, mais aussi pour compiler chaque fichier (C et assembleur) en fichiers objets (`.rel`) pour ensuite passer au stade de l'édition de liens et produire le binaire. Ceci est important, car c'est le seul moyen pour que toutes les références soient dynamiquement mises à jour. Notez également que l'option `--code-loc` est passée à `0x001C` puisque notre crt0 s'est étoffé.

La cible `paddump` du `Makefile` (anciennement `dumppad`) produira la variable à utiliser dans le croquis Arduino. Celui-ci peut être identique à celui de l'article précédent, mais vous en trouverez une version légèrement modifiée (et accélérée) dans le dépôt GitHub du magazine, en compagnie, bien entendu, de tous les autres fichiers du projet. Cette version fait l'impasse sur l'affichage du déroulement de l'exécution en mémoire pour se concentrer sur les données transitant depuis et vers le 16550.

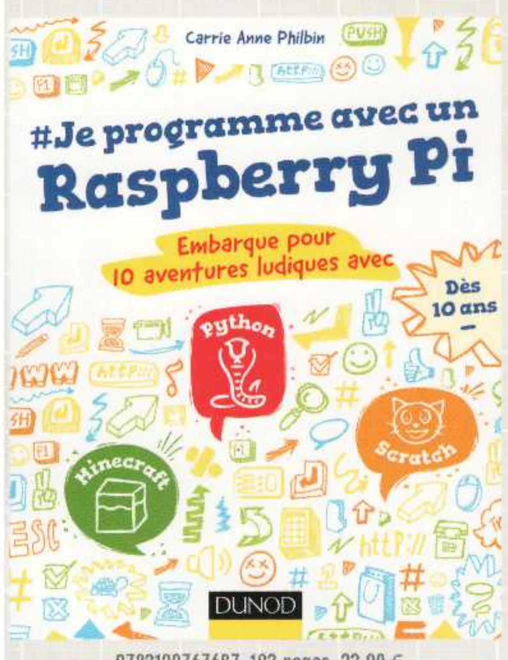
LE MOT DE LA FIN

J'ai parfaitement conscience que cette étape est un « gros morceau » condensé en bien peu de pages, mais si vous avez suivi les articles précédents, vous devez disposer des bases nécessaires pour sauter le pas sans encombre et sans trop de maux de tête. L'approche que j'ai utilisée ici n'est pas la seule possible, mais c'est la plus simple. Une autre consisterait à faire apparaître les 8 registres du 16550 comme une partie de mémoire et donc d'avoir ROM, RAM et ces registres dans le même espace d'adressage. C'est ce que fait, par exemple, le Commodore 64 avec les 29 registres du SID 6581/8580 (cf. article sur le sujet dans ce numéro). Ceci nécessite davantage de circuits logiques annexes afin de lier certaines lignes du bus d'adresse aux signaux *Chip Select*, mais facilite la vie au programmeur par la suite.

La prochaine étape du projet n'est pas encore clairement définie à ce stade. Comme vous pouvez le constater sur les illustrations de l'article, le projet commence à ressembler à un nid de souris des moissons et il va falloir envisager sérieusement une approche différente de la platine à essais. Mais l'envie est également assez grande de brûler cette étape et de mettre en œuvre une vraie RAM statique (SRAM) pour reléguer l'Arduino au rang de simple moniteur d'exécution. Nous verrons... **DB**

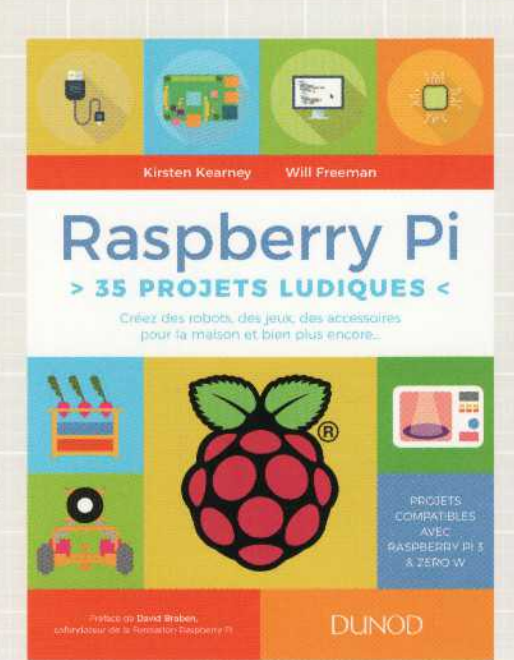
TOUS MAKERS !

RÉVÉLEZ VOTRE POTENTIEL PAR LA CRÉATION



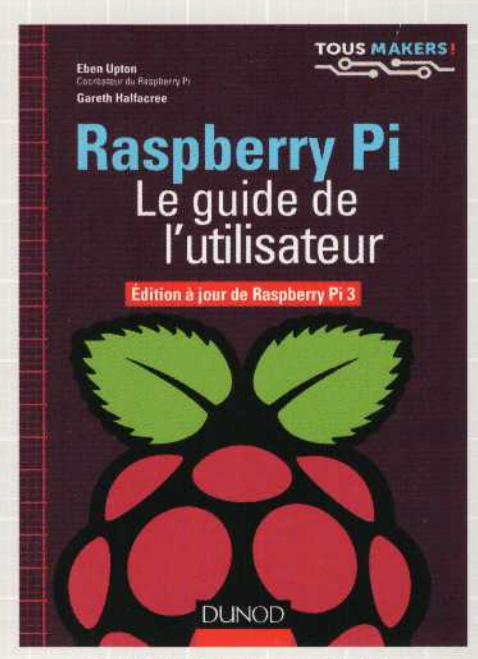
9782100767687, 192 pages, 22,90 €

Pour les jeunes programmeurs en herbe qui souhaitent apprendre à coder et créer leurs propres jeux avec un Raspberry Pi.



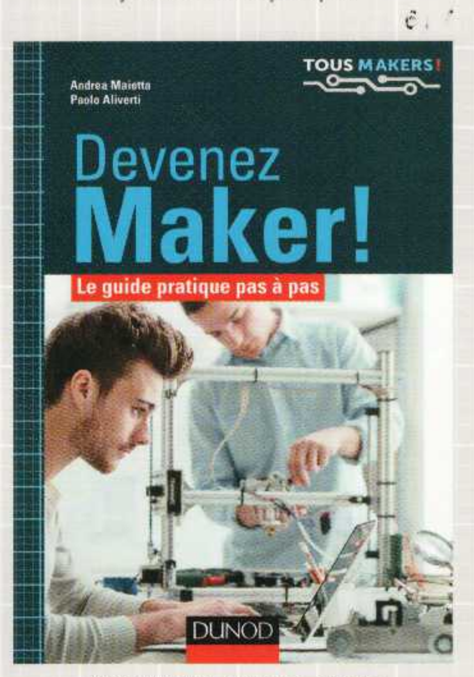
9782100775859, 224 pages , 24,90 €

Innovez avec votre Raspberry Pi : les conseils de véritables pros du Pi pour réussir des projets innovants et ludiques.



9782100762262, 288 pages, 26,90 €

Écrit par le co-créateur du Raspberry Pi, cet ouvrage donne toutes les clés pour tirer le meilleur parti du nano-ordinateur révolutionnaire.



9782100762934, 304 pages, 24,90 €

Comment transformer vos idées en projets concrets ? Ce livre vous accompagne dans la réalisation de vos premières créations.



9782100758487, 240 pages, 27 €

Le compagnon idéal pour vous guider pas à pas tout au long de la construction de votre drone.



9782100738106, 176 pages, 23 €

Un guide pour construire votre machine CNC, idéale pour la gravure et le perçage des circuits imprimé les maquettistes et les modélistes.

TOUT LE CATALOGUE SUR WWW.DUNOD.COM

